Hacker de jogos Faculdade

Guia de Compreensão para Iniciantes
Técnica de hacking de jogos

Foi criado com base em materiais publicados originalmente em https://gamihacking.ACDEMY e https://gamihacking.ACDEMY entre 2019 e 2021.

Fundado em 2021.

Este livro é distribuído gratuitamente. Se você quiser apoiar trabalhos futuros, sinta-se à vontade para doar:

https://www.buymeacoffee.com/gamehackingacad

Há algum problema ou um problema de notificação? Sinta-se à vontade para entrar em contato comigo por e-mail ou Twitter:

Attitathedud@gmail.com

https://twitter.com/GameHackingAcad

Fotos de computador tiradas por Luke Hodde no Unsplash

Licenciado sob a Licença Pública Internacional Creative Commons Atribuição-NonCommercial 4.0.

Catálogo

Catálogo 3

Introdução 7

Recursos externos 8

Conhecimento básico 9

1.1 Noções básicas de informática 10

1.2 Noções básicas do jogo 18

1.3 Fundamentos de hackers 24

1.4 Instalar uma máquina virtual de laboratório 26

1.5 hackers de memória 29

Depuração e reversão 37

2.1 Fundamentos de depuração 38

2.2 Fundamentos de reversão 41

2.3 Alterar o código do jogo 45

2.4 Código de ré 56

2.5 Código Caverna 68

2.6 Use Caverna do Código 72

2.7 alocação de memória dinâmica 2.8 supera DMA 78 82

Programação 90

3.1 Noções básicas de programação 3.2 Hacker externo 91 95

3.3 DLL Memory Hacker 110
3.4 Code Cave e DLL 3. 122
5 Texto impresso 131

Hacker RTS 142

4.1 STATACK 143

4.2 Hacker de Mapas 151

4.3 Macro Robôs 160

Hacker FPS 175

5.1 Fundamentos 3D 5 176
.2 Wallhack (Memória) 183
5.3 Wallhack (OpenGL) 196
5.4 Chams (OpenGL) 5 217
.5 Trigerbot 221

5.6 Amor 230

5.7 Sem recuo 253
5.8 Radar 260
Hacker 5.9 ESP 266

5.10 jogos 298
multijogador 312

6.1 Fundamentos de Jogos 313
Multijogador 6.2 Análise de Pacotes 317

6.3 Agrupamento Inverso 331

6.4 Criar clientes externos 6.5 345
tráfego TCP proxy 356

Desenvolvimento de ferramentas 365

7.1 Injetor DLL 366

7.2 Scanner de modo 7. 372
3 Scanner de memória 7. 380
4 Scanner de montagem 389

7.5 Depurador 7.6 403
Gravador de chamadas 409

Apêndice A 415

Script de configuração VM do laboratório A1 416

A2 Wisno Hacker de ouro externo 416
A3 Wisno Hacker de ouro interno 418
A4 Wisno Código Cave, etc. 419

A. 5 Vesnos Starhak 421

A.6 Hacker de mapas Wesnoth 424

Macro robô A.7 Wyrmsun 425

A8 Urban Horror Memory Wallhack 428
A9 Urban Horror OpenGL Wallhack 430
A10 Urban Horror OpenGL Chams 433
A11 Ataque Cube Robô 437

Aimbot Cubo de Ataque A12 439

A13 Cubo de Ataque Sem 442
Recuperação A14 Cubo de Ataque ESP 443

A.15 Cubo de Ataque Multihack 448
A.16 Wesnoth Robô Multijogador 480

A.17 Wesnoth Chatbot 483
A.18 Wesnoth A.19 487
DLL injetor 491

Scanner de padrão A20 493
, scanner de memória 495
A21, inversor A22 499

Depurador A23, gravador 503
de chamadas A24 506

Introdução

Os jogos de hacking exigem uma combinação única de habilidades de reversão, gerenciamento de memória, rede e segurança. Embora os hackers éticos tenham se tornado populares, os hackers de jogos ainda ocupam uma posição pequena em comunidades seguras mais amplas. Embora ele possa não ter o mesmo apelo de título ou vazamento de dados que o Chrome 0day, a sensação única de criar um aimbot funcional para o jogo e depois destruir um servidor com ele é difícil de replicar em qualquer outra mídia.

Quando comecei a aprender hacking de jogos, alguns anos atrás, os recursos estavam espalhados por vários sites e eram muito escassos. Normalmente, você encontrará um pedaço de código vinculado a um site danificado. Você então pesquisará em alguns fóruns que contêm conteúdo de sites quebrados em uma postagem e juntará essas informações. Embora valha a pena uma pesquisa completa, é um receptor de tempo enorme. Hoje em dia, há vários lugares onde você pode encontrar várias informações sobre hackers de jogos . Você pode encontrar códigos de modelo para quase qualquer mecanismo, bem como deslocamentos de memória para qualquer estrutura com a qual você possa se preocupar.

No entanto, uma área em que todas as informações ainda não atendem adequadamente hoje é a compensação dos conceitos e fundamentos por trás disso. Espero que este livro ajude a preencher essas lacunas.

-attilathedud

Recursos externos

Esta é uma lista de todos os recursos externos usados neste livro, como ferramentas e jogos. Eles são classificados por aparência. Isso ajudará se você planeja ler este livro em um lugar sem internet.

- •VirtualBox (https://www.virtualbox.org/wiki/Downloads)
- •Máquina virtual de avaliação de Windows 10 (https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/)
- • Mecanismo de trapaça (https://cheatengine.org/)
- •x64dbg(https://x64dbg.com/)
- • Batalha de Vesnos 1.14.9 (https://www.wesnoth.org/)
- •Edição de Comunidade Visual 2019 (https://Visual Studio.microsof.com/vs/Comunidade/)
- Wyrmsun(https://store.steampowered.com/app/370070/Wyrmsun)
- Terror urbano 4.3.4. (https://www.urbanterror.info/)
- Cubo de Ataque 1.2.0.2 (https://assault.cubers.net/)
- wireshark (https://www.wireshark.org/)
- cmder(https://cmder.net/)
- zlib (https://zlib.net/)
- Batalha de Vesnos 1.14.12 (https://www.wesnoth.org/)

Parte 1

Conhecimento básico

1.1 Computador Fundamentos

1.1.1 Componentes de computador

Um computador típico tem várias peças conectadas. Os mais importantes são:

- Disco rígido
- RAM
- Placa de vídeo
- Placa-mãe
- CPU

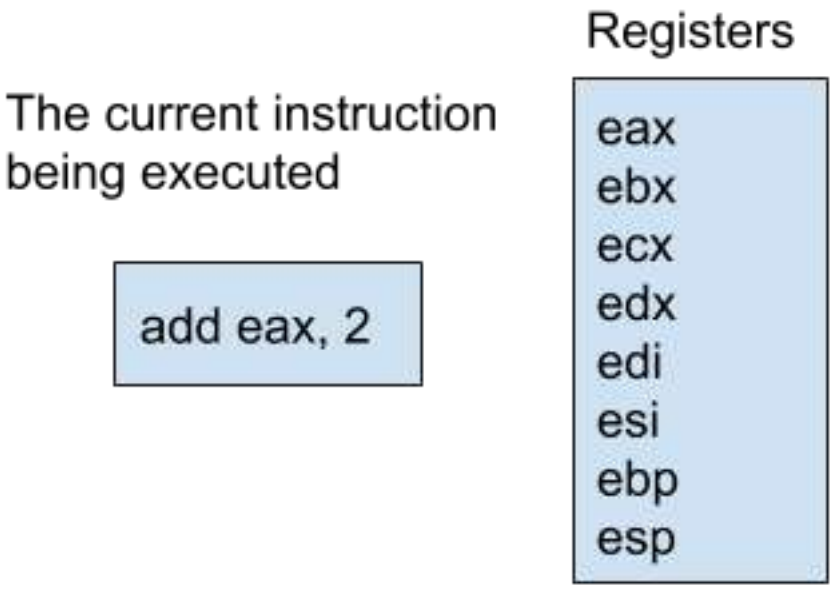
Se você quiser remover a lateral do seu computador de mesa, as peças poderão ser colocadas na seguinte configuração:

Para nossos propósitos, apresentaremos brevemente apenas os quatro primeiros componentes e, em seguida, nos concentraremos na CPU na próxima seção:

- O disco rígido é responsável por armazenar arquivos grandes, como fotos, arquivos executáveis ou arquivos de sistema.
- RAM armazena dados que exigem acesso rápido. Os dados são carregados do disco rígido para o RAM.
- A placa de vídeo é responsável por exibir gráficos no monitor.
- A placa-mãe vincula todos esses componentes e permite que eles se comuniquem.

1.1.2 CPU

A CPU é o cérebro do computador. É responsável por executar instruções. Essas instruções são simples e variam de acordo com a arquitetura. Por exemplo, uma instrução pode somar dois números. Para acelerar a execução, a CPU tem várias áreas especiais para armazenar e modificar dados. Eles são chamados de registros.



1.1.3 Explicação

Todos os programas de computador são compostos por uma série de instruções. Como discutimos acima, uma instrução é simples e geralmente faz apenas uma coisa. Por exemplo, aqui estão algumas instruções na maioria das arquiteturas:

Some dois números.
Subtraia dois números.
Compare os dois números.
Mova números para uma parte da memória (RAM).
• Vá para outro código.

Programas de computador são desenvolvidos combinando essas instruções simples. Por exemplo, uma calculadora simples pode ser a seguinte:

```
mov eax, 5
mov ebx, 4
add eax, ebx
```

A primeira instrução (mov) move o valor 5 para o registro eax. O segundo move o valor 4 para o registro ebx. A instrução add então adiciona eax e ebx e coloca o resultado de volta em eax.

1.1.4 Programa de Computador

Programas de computador são um conjunto de instruções. O programa é responsável por receber um valor (entrada) e, em seguida, gerar um valor (saída) com base no valor recebido.

Por exemplo, um programa simples pode usar um número como entrada, adicionar 1 ao número e então movê-lo para a saída. Pode parecer:

```
mov eax, entrada
adicionada eax, 1
Saída mov, eax
```

Um programa mais complexo terá muitos programas simples como este "interno". Nesse caso, esses programas internos simples são chamados de funções. A função é como um programa, aceitando entradas e gerando saídas. Por exemplo, podemos fazer do nosso programa anterior uma função para fazer a mesma coisa. Pode parecer:

```
Função adicionada (
entrada): mov eax,
entrada adicionada eax, 1

Saída mov, eax
```

Também podemos criar outra função para executar uma operação semelhante. Por exemplo, podemos escrever uma função para subtrair um número em 1:

```
Subtração de função (
entrada): mov eax, entrada
sub eax, saída de 1
mov, eax
```

Essas duas funções (adição e subtração) podem ser usadas para criar um programa mais complexo. Este novo programa aceita um número e o aumenta ou diminui. Requer duas entradas:

1. Um número
2. Uma operação matemática, neste caso, adição (+) ou subtração (-)

O novo programa será mais longo e será executado de duas maneiras diferentes. Isso será explicado após o código:

```
Função adicionada (
entrada): mov eax, entrada

Adicionar eax, 1
Saída mov, eax

Subtração de função (
entrada): mov eax, entrada
sub eax, saída de 1
mov, operação eax
cmp, '-'
```

```
je Subtracta_Number
add (number)
Saída

Subtracta_Number:
Subtrair (números)
para sair
```

Este código tem duas funções no topo. Como discutimos, esses aceitam uma entrada e então adicionam ou subtraem 1 da entrada para gerar uma saída. A instrução cmp compara os dois valores. Neste caso, ele compara o tipo de operação recebida como entrada com o valor codificado no programa -. Se esses valores forem iguais, vamos para (ou saltamos para) outro código (je=jump se forem iguais).

Se a operação for igual a -, vamos para o código que subtrai 1 do número. Caso contrário, continuamos com o programa e adicionamos o número 1 antes de sair.

Comparar números e depois saltar para códigos diferentes com base em seus valores é chamado de ramificação. Os ramos são componentes essenciais no design de programas complexos que podem responder a diferentes entradas. Por exemplo, um jogo geralmente tem um ramo em todas as direções que o jogador pode se mover.

1.1.5 binário, decimal e hexadecimal

Fundamentalmente falando, a CPU é um circuito. O circuito tem corrente fluindo através (ligado) ou não (desligado). Ambos os estados podem ser representados usando um sistema binário (ou com base em 2). Em um sistema com cardinalidade 2, há dois valores possíveis: 0 e 1. O número binário de exemplo é 1101.

Estamos familiarizados com o sistema numérico decimal (ou baseado em 10), que tem 10 valores possíveis: 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9. O número decimal de exemplo é 126. Este número pode ser expresso em um formato mais claro como:

$$(1*10^2)+(2*10^1)+(6*10)^0$$

Podemos representar o número binário acima (1101) no mesmo formato. No entanto, substituiremos 10 por 2 porque mudaremos do decimal para o binário:

$$(1*2^3)+(1*2^2)+(0*2^1)+(1*2^0)$$

Quando os números binários precisam representar valores maiores, eles rapidamente se tornam desajeitados. Por exemplo, a representação binária do número decimal 250 é 11111010.

Para representar esses números binários maiores, números hexadecimal (base 16) são geralmente usados em cálculos. Os números hexadecimal geralmente são prefixados com o identificador 0x E têm dezesseis valores possíveis: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E E F. O número hexadecimal de exemplo é 0xA1D.

1.1.6 Linguagem de programação

As instruções são representadas como números, assim como todos os outros dados em um computador. Esses números são chamados de códigos de operação, geralmente simplesmente chamados de códigos de operação. O código de operação varia de acordo com a arquitetura. A CPU conhece cada código de operação e o que fazer ao encontrar cada código de operação. Os códigos de operação geralmente são expressos em formato hexadecimal. Por exemplo, se uma CPU Intel encontrar 0xE9, ela sabe que precisa executar uma instrução jmp (salto).

Os primeiros computadores exigiam o uso de códigos de operação para escrever programas. Isso é obviamente difícil de fazer, especialmente para procedimentos mais complexos. Em seguida, uma variante da linguagem de assembleia é usada para permitir a escrita de instruções. Eles parecem semelhantes aos exemplos que escrevemos acima. Linguagens de montagem são mais fáceis de ler do que opercode, mas ainda são difíceis de usá-las para desenvolver programas complexos.

Para melhorar a experiência de desenvolvimento, vários idiomas de alto nível, como FORTRAN, C e C++, foram desenvolvidos. Essas línguas são fáceis de ler e introduzem operações de controle de fluxo, como condições se e se. Por exemplo, abaixo está nosso programa incremental/decrescente na linguagem C. Na linguagem C, int se refere a números inteiros, ou números inteiros (por exemplo, -1, 0, 1 ou 2).

```
int add {return int+1;

}

int subtract{int int Int-1
;
}

Se (operatização=='-'){
subtract (num);
```

```
}
else{add(number
);}
```

Todas essas línguas de alto nível são compiladas em línguas de assembleia. O programa de montagem tradicional então converte a montagem em um código de operação que a CPU pode entender.

1.1.7 Sistema Operacional

A comunicação de programação com hardware é um processo demorado e difícil. Para executar nosso programa incremental/decrescente, também temos que escrever código para processar cliques no teclado, exibir gráficos no monitor, construir conjuntos de caracteres para que possamos representar letras e números e nos comunicar com RAM e discos rígidos. Para facilitar o desenvolvimento do programa, foi criado um sistema operacional. Eles contêm código que já pode lidar com essas funções de hardware. Eles também têm várias funções padrão comumente usadas, como copiar dados de um local para outro.

Os três principais sistemas operacionais ainda em uso hoje incluem Windows, Linux e MacOS. Tudo isso tem bibliotecas e métodos diferentes para se comunicar com hardware . É por isso que programas escritos para Windows não podem ser executados no Linux.

1.1.8 Aplicação

O sistema operacional precisa de uma maneira de determinar como os usuários processam os dados quando os selecionam. Se os dados forem uma foto, o sistema operacional quer chamar um aplicativo específico, como o Paint, para visualizar a foto. Da mesma forma, se os dados em si forem um aplicativo, o sistema operacional precisa transferi-los para a CPU para execução.

Cada sistema operacional processa a execução de uma maneira única. No Linux, permissões executáveis especiais são definidas para arquivos normais. No Windows, os aplicativos são formatados de uma maneira especial que o Windows sabe como analisar. Isso é chamado de PE ou formato executável portátil. O formato PE tem várias partes, como a parte de texto que armazena o código do programa e a parte de dados que armazena a variável.

1.1.9 Jogo

Tudo isso foi resolvido e finalmente podemos discutir o jogo. Jogos são apenas aplicativos. No Windows, eles são formatados no formato PE, o mesmo que qualquer outro aplicativo. Eles contêm uma seção .text que armazena o código do programa que consiste em códigos de operações. A CPU então executa esses códigos de operação, e o sistema operacional exibe os gráficos gerados e processa entradas, como teclas.

1.2 Jogo Fundamentos

1.2.1 Parte do jogo

Embora os jogos sejam aplicativos, eles são complexos e consistem em várias partes. Alguns deles incluem:

- Gráficos
- som
- Entr
- ada
- Física
- Lógica do jogo

Devido à complexidade de cada parte, a maioria dos jogos usa recursos externos para essas partes. Essas funções externas são combinadas em uma chamada biblioteca. Outros programas então usam a biblioteca para reduzir a quantidade de código escrito. Por exemplo, para desenhar imagens e formas na tela, a maioria dos jogos usa bibliotecas DirectX ou OpenGL. Para certos tipos de hackers, é importante identificar as bibliotecas que estão sendo usadas. wallhack é um tipo de hacker que permite que hackers vejam outros jogadores através de paredes fortes. Uma maneira de escrever o wallhack é modificar a biblioteca gráfica do jogo. Tanto o OpenGL quanto o DirectX são vulneráveis a esse tipo de hacker, mas cada um requer uma abordagem diferente.

Para a maioria dos hackers, modificaremos a lógica do jogo. Esta é a seção responsável pela explicação da jogabilidade do jogo. Por exemplo, a lógica do jogo controlará a altura do personagem salta ou quanto dinheiro o jogador recebe. Ao mudar esse código, podemos pular o máximo que quisermos ou ganhar dinheiro ilimitado.

1.2.2 Estrutura do Jogo

Como todo código de computador, a lógica do jogo é composta de instruções. Devido à complexidade dos jogos, eles geralmente são escritos e compilados em linguagens de alto nível. Para ataques de hackers mais complexos, geralmente é necessário entender a estrutura geral do código original.

A maioria dos jogos tem dois recursos principais:

- Configurações
- Circuito principal

As configurações são executadas no início do jogo. Ele é responsável por carregar imagens, sons e outros arquivos grandes do disco rígido e colocá-los no RAM. O loop principal é um tipo especial de função que funciona até o jogador sair. É responsável por processar entradas, reproduzir sons, atualizar telas, etc. O exemplo do loop principal pode ser o seguinte:

```
Função main_loop(){handle
_input();
update_score();
Play_SOUND_EFCTS(); draw_
screen ();
}
```

Todas essas funções chamam outras funções. Por exemplo, a função handle_ input pode ser a seguinte:

```
Função handle_input(){
Se(keydown==LEFT){
UPDate_Player_Posição (GO_LEFT);}

Esse if (keydown==RIGHT){UPDate_Player_
Posição (GO_RIGHT);}

}
```

O programa de cada jogo é diferente. Alguns jogos podem priorizar atualizações gráficas antes de processar entradas. No entanto, todos os jogos têm algum tipo de loop principal.

1.2.3 Dados e Classe

Quaisquer dados que possam ser atualizados no jogo são armazenados em variáveis. Isso inclui a pontuação, posição ou dinheiro do jogador. Essas variáveis são declaradas no código. Exemplos de definições de variáveis na linguagem C podem ser mostrados abaixo:

```
int money=0;
```

Este código declara a variável de dinheiro como um número inteiro. Como aprendemos no capítulo anterior, o valor inteiro na linguagem C é um número inteiro (por exemplo, 1, 2 ou 3). Imagine se tivéssemos que rastrear o dinheiro de alguns jogadores. Uma abordagem é ter várias declarações, conforme mostrado abaixo:

```
int Money 1=0;
int Money 2=0;
int money3=0;
int Money 4=0;
```

Uma desvantagem dessa abordagem é que ela é difícil de manter à medida que o jogo se torna maior e mais complexo. Por exemplo, para escrever um código que aumenta o dinheiro de cada jogador em 1, temos que atualizar manualmente cada variável:

```
Função increase_money(){
money 1=money 1+1; money 2=
money 2+1;
...}
```

Se adicionarmos outro jogador, teremos que atualizar o código de cada segmento que muda o dinheiro do jogador. Uma abordagem melhor é declarar esses valores na lista. Podemos então usar uma instrução chamada loop para atravessar cada item na lista. Isso é o que se chama de iteração. Na linguagem C, as listas são geralmente implementadas usando o chamado array. Para nossos propósitos, você pode presumir que lista e matriz são sinônimos. Um ciclo na linguagem C é chamado de ciclo for. O loop For é dividido em três partes: o valor inicial, o valor final e como atualizar o valor após cada iteração. O exemplo do código anterior pode ser o seguinte:

```
int money[10]={0};
int Currente_Players=4;

Função increase_money(){
for (int i=0; i<current_players;i++) {dinheiro[i
]=dinheiro[i]+1;
}}
```

Agora precisamos apenas atualizar a variável current_players para adicionar suporte a outro player.

Para facilitar o desenvolvimento de aplicativos complexos, os desenvolvedores geralmente usam modelos de programação chamados programação orientada a objetos (OOP). Em OOPs, variáveis e funções são combinadas em conjuntos chamados classes. As classes geralmente são autocontidas. Por exemplo, muitos jogos têm um tipo de jogador. Esta classe conterá várias variáveis, como localização, nome ou dinheiro do jogador. Essas variáveis na classe são chamadas de membros. A classe também incluirá funções para modificar esses membros. Um exemplo da classe Player pode ser o seguinte:

```
Jogador {
moeda internacional;
Nome da string;

Função increase_money(){money=
money+1;}

}
```

Os jogos geralmente incluem uma lista de cursos. Por exemplo, o jogo Thor Hammer 3 tem uma matriz de todos os jogadores atualmente conectados ao servidor. Cada jogador tem sua própria profissão de jogador no jogo. Para calcular a tela de pontuação, o jogo verifica cada jogador na lista para ver quantas vezes ele matou.

1.2.4 Memória

O jogo tem muitos recursos grandes, como imagens e som. Eles devem ser carregados do disco rígido, geralmente durante a fase de configuração do jogo. Uma vez carregados, eles são colocados

Em RAM, bem como código e dados do jogo. Como os jogos são muito grandes, eles precisam carregar constantemente dados diferentes do RAM no registro para operação. Esse carregamento geralmente é feito pelo comando mov. Este comando move parte da memória para o registro. A função de exemplo increase_money executada pela CPU pode ser a seguinte:

```
Função increase_money:
mov eax, 0x12345678
Adicionar eax, 1
mov 0x12345678, eax
```

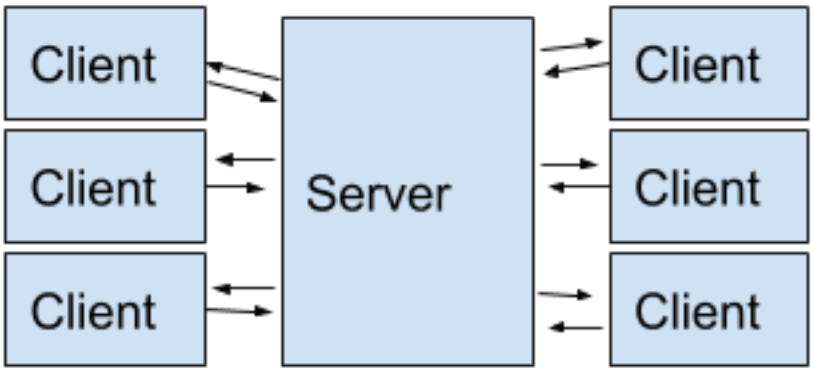
Neste exemplo, usamos 0x12345678 como a posição do dinheiro do jogador no RAM . A maioria dos jogos tem essa estrutura, mas em locais diferentes. Para jogos mais complexos, essas posições serão baseadas em outras posições. Se nosso jogo tiver uma classe Player, o código increase_money executado pela CPU precisará usar a localização da classe do Player para pesquisar moeda.

```
Função increase_money:
mov ebx, 0x12345670 mov
ebx, ebx+8 Adicionar ebx,
1
mov ebx+8, eax
```

Nesse caso, a CPU deve compensar a posição do dinheiro com base na posição da classe do jogador.

1.2.5 Cliente multipessoal

Os jogos multijogador permitem que vários jogadores interajam entre si. Para isso, os jogos multijogador utilizam clientes e servidores. Um exemplo do modelo cliente -servidor é mostrado abaixo:



O cliente representa a cópia do jogo de cada jogador e contém todas as informações sobre o jogo local. Por exemplo, cada cliente incluirá o dinheiro do jogador. Quando um jogador faz com que uma ação mude seu dinheiro, o cliente é responsável por enviar a atualização ao servidor.

As informações também podem ser enviadas em ambas as direções. Um exemplo disso é o movimento do jogador. Um cliente informará ao servidor que o jogador moveu sua localização. O servidor então informa a todos os outros clientes que atualizem suas localizações relevantes para clientes móveis.

1.2.6 Servidor multijogador

Embora o cliente represente a cópia do jogo de cada jogador, o servidor garante que todos os clientes conectados estejam jogando a mesma cópia. Os servidores geralmente os restringem de aceitar alterações do cliente. Por exemplo, imagine que escrevemos um hacker para mudar nosso dinheiro no jogo. Se for um jogo multijogador, o servidor rejeitará nossas alterações. É por isso que hackers single-player geralmente não funcionam em jogos multijogador.

Discutiremos mais detalhadamente os princípios básicos dos jogos multijogador em capítulos subsequentes.

1.3 Ataque de hackers

Fundamentos

1.3.1 Etapas de ataque de hackers

Todas as atividades de hacking incluem modificar a memória no jogo. Escrever qualquer hack envolve quatro etapas principais:

1. Determine o que você quer mudar.
2. Entenda qual memória você precisa localizar.
3. Encontre memória no jogo.
4. Mude sua memória.

Essas etapas se aplicam a qualquer ataque hacker, independentemente da complexidade.

1.3.2 Identificação

O primeiro passo para qualquer ataque de hacker é determinar o que você quer fazer. Hackers diferentes exigem métodos diferentes. Por exemplo, modificar o dinheiro do jogador exigirá modificações de memória em variáveis, enquanto ver outros jogadores através da parede exigirá modificações de memória no código do jogo.

1.3.3 Compreensão

Para modificar a memória, precisamos encontrá-la. Antes de tentar localizá-lo, precisamos entender que tipo de memória precisamos localizar. Em alguns casos, a memória que você deseja modificar será uma variável. Em outros casos, você gostaria de modificar a maior parte do código. Existem três tipos principais de modificações:

Variáveis, como modificar o valor do dinheiro do jogador
• Código, como modificar como as paredes são desenhadas
• Arquivos, como modificar itens salvos no estoque

1.3.4 Posicionamento

Depois de saber o que quer mudar e onde procurar, você pode começar a procurar. Para alguns hackers, isso pode envolver pesquisar por memória usando uma ferramenta chamada scanner de memória. Para outros, isso pode incluir o uso de uma ferramenta chamada depurador para visualizar o código do jogo. Dependendo do jogo e do método, esse pode ser um processo demorado.

1.3.5 Alterações

Depois de encontrar a memória, o último passo é alterá-la. Inicialmente, isso significava modificar manualmente a memória usando um scanner ou depurador de memória. Depois de verificar que isso é viável, você pode escrever um programa para alterá-lo automaticamente para você.

1.4 Instalar uma máquina virtual de laboratório

1.4.1 Visão geral

Ao realizar qualquer tipo de ataque de hacker, é melhor separar sua máquina pessoal da máquina de hacker. Uma maneira simples de fazer isso é usar uma máquina virtual. Neste capítulo, configuraremos uma máquina virtual de hacker de jogos que executa o Windows 10. Este também será o ambiente que usaremos em todos os outros capítulos.

Devido aos requisitos de hardware do jogo, você frequentemente descobre que é impossível executar um jogo em um VM. Nesses casos, uma opção é criar outra parte no disco rígido chamada partição. Você pode então instalar o Windows nesta partição separada e mantê-lo para hackers do jogo.

1.4.2 VirtualBox

Neste livro, usaremos um gerenciador de máquinas virtuais chamado VirtualBox. Este software permite que você execute e gerencie máquinas virtuais. Você pode baixá-lo aqui.

1.4.3 Máquina virtual

Em seguida, precisamos de uma máquina virtual. Como a maioria dos jogos é lançada para o Windows, usaremos a máquina virtual Windows 10. A Microsoft lançou a máquina virtual Windows 10 gratuita para testar a versão antiga do Internet Explorer. Estes são totalmente legais, mas têm um limite de ativação de 90 dias. Para nossos fins, baixaremos uma imagem VirtualBox. Você pode baixar imagens aqui. Observe que essas máquinas virtuais têm aproximadamente 7 GB de tamanho. Após baixar a imagem, o arquivo OVF é extraído do arquivo. As imagens VirtualBox usam extensões de OVA e OVF. Essas imagens contêm cópias salvas de computadores pré-configurados. Depois que configuramos nossa máquina, criaremos um instantâneo dela. Isso nos permitirá jogar fora a máquina antiga e criar uma nova cópia quando ela expirar. Nunca guarde notas ou pertences pessoais na máquina virtual.

Abra o VirtualBox e importe o OVF baixado. Mantenha todas as opções como padrões padrão.

Após importar a máquina virtual, inicie-a. O Windows 10 será ativado e enviará uma tela de login. A senha do usuário é "Passw0rd!", sem aspas.

1.4.4 Ferramentas

Usaremos um script Boxstarter para instalar algumas ferramentas de hacking de jogos. O Boxstarter é um conjunto de utilidades que permite que você escreva scripts e recrie instalações. Neste caso, usaremos isso para instalar buscadores de memória e depuradores. Na máquina virtual, abra Powershell e execute os dois comandos a seguir:

```
.{iWr-USB https://boxstarter.org/Bootstrapar.ps1}|iex; Obter-
BoxStarter-Força

Install-BoxStarterpackage-package https://
Raw.githuubusContente.com/gamihachingacademe/vmsetup/master/
vmsetup/txt-disserbooks
```

O arquivo vmsetup.txt também é fornecido no Apêndice A. Se você estiver usando a versão local, execute o segundo comando a seguir em vez da versão acima:

```
Install-BoxStarterpackage-package c:\
location\of\vmsetup txt-disablerboard books
```

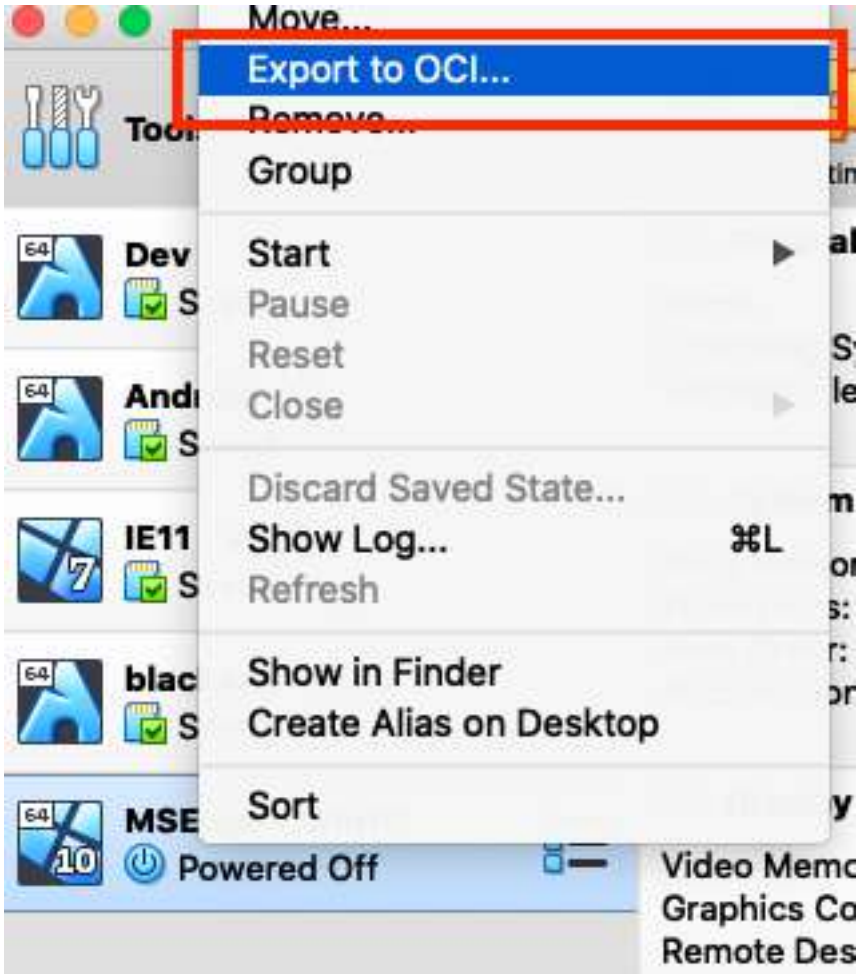
O primeiro comando foi instalar o BoxStarter, retirado do site deles. O segundo comando é um script de instalação que habilita algumas opções de pasta e instala três programas:

1. Mecanismo de trapaça, um scanner de memória
2. x64dbg, um depurador
3. Gerente de Embalagens Chocolate

Quando você encontrar mais ferramentas, você deve criar seus próprios scripts e adicioná-los.

1.4.5 VM clonado

Agora que configuramos o ambiente, vamos criar um clone. Isso nos permitirá recriar a máquina virtual usando todas as ferramentas instaladas. Desligue completamente a máquina virtual e selecione exportar para o OCI. Guarde todos os valores padrão e comece a exportar.



Isso criará um OVA, semelhante ao OVF que baixamos originalmente. Se destruirmos nosso ambiente, podemos simplesmente excluí- lo e importar esta nova OVA. Ele já instalou todas as nossas ferramentas, então não precisamos perder tempo reinstalando-as.

1.5 hackers de memória

1.5.1 Objetivo

Para nosso primeiro ataque de hackers, vamos mirar em um jogo chamado Wisnoth Battle. Este é um jogo gratuito de código aberto, sem mecanismo antifraude. A maioria dos capítulos deste livro será voltada especificamente para a versão 1.14.9. Para instalá-lo em nosso computador virtual, abra o Powershell como administrador e execute os seguintes comandos:

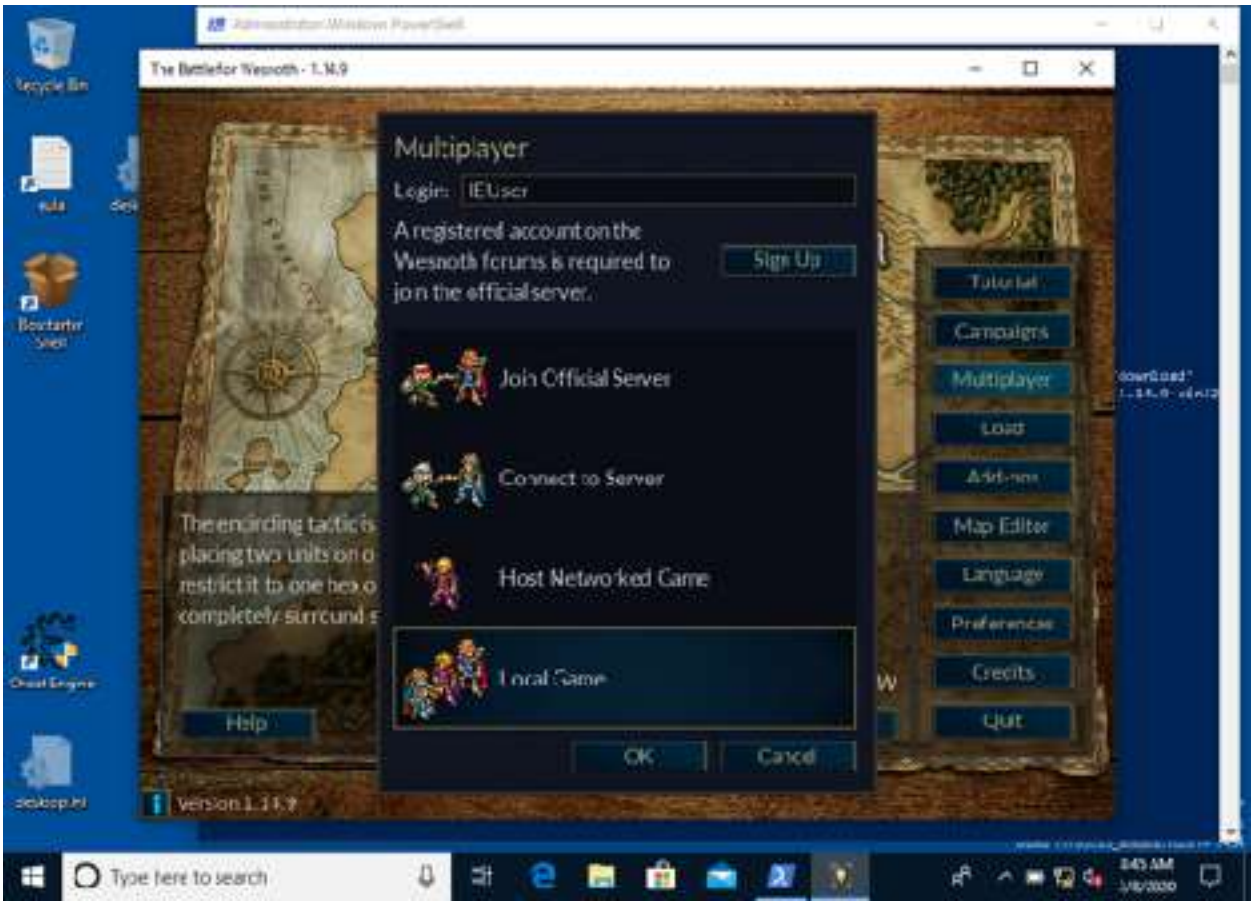
```
choco Instale wesnoth - Versão = 1.14.9 -y
```

Isso usará chocolate para instalar a Batalha de Vistanos. Após a instalação, abra o jogo e verifique se ele está funcionando corretamente. Em seguida, entre no menu de preferências e mude o modo de vídeo do jogo para o modo janela. Por fim, jogue tarefas tutoriais para entender como o jogo funciona.

1.5.2 Identificação

O objetivo do nosso ataque de hackers desta vez é mudar nosso ouro. Os jogadores usam ouro para comprar tropas no jogo, então quanto mais ouro temos, mais tropas podemos comprar. Faremos isso usando uma ferramenta chamada Cheating Engine, que nos permite escanear e modificar a memória do jogo. Para isso, usaremos os passos que aprendemos no Capítulo 1.3.

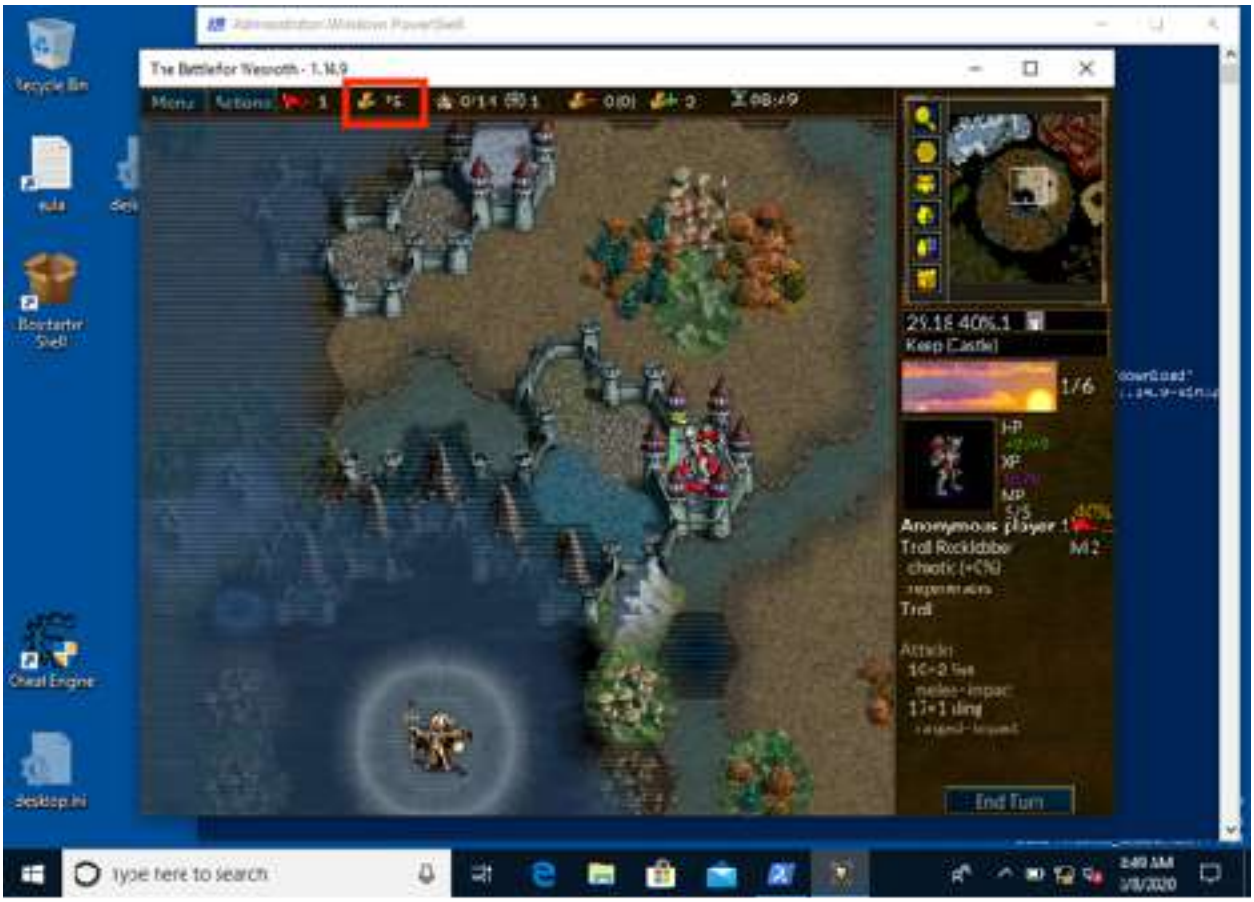
Os jogadores só podem obter moedas de ouro enquanto jogam uma cena. Para criar uma cena, selecione jogos multijogador e depois selecione jogos locais.



Guarde os valores padrão das configurações restantes e comece o jogo.

1.5.3 Compreensão

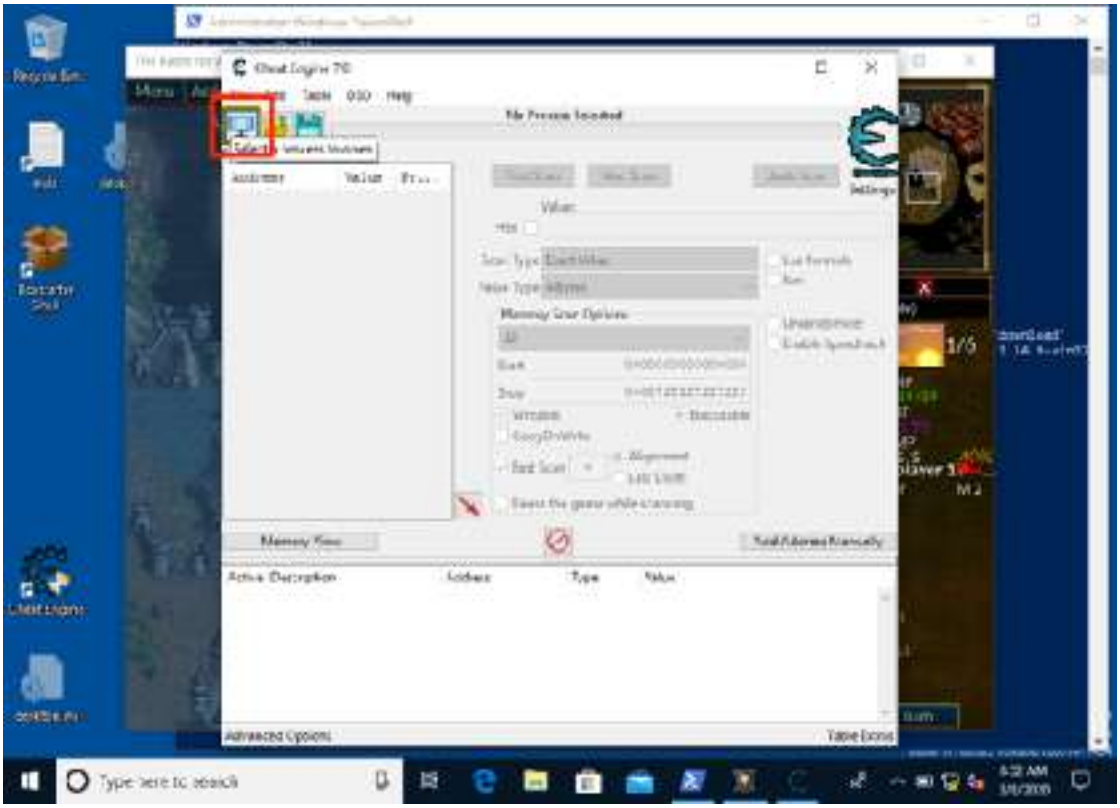
Neste jogo, o ouro do jogador é armazenado em uma variável. Essa variável é referida pelo código do jogo.



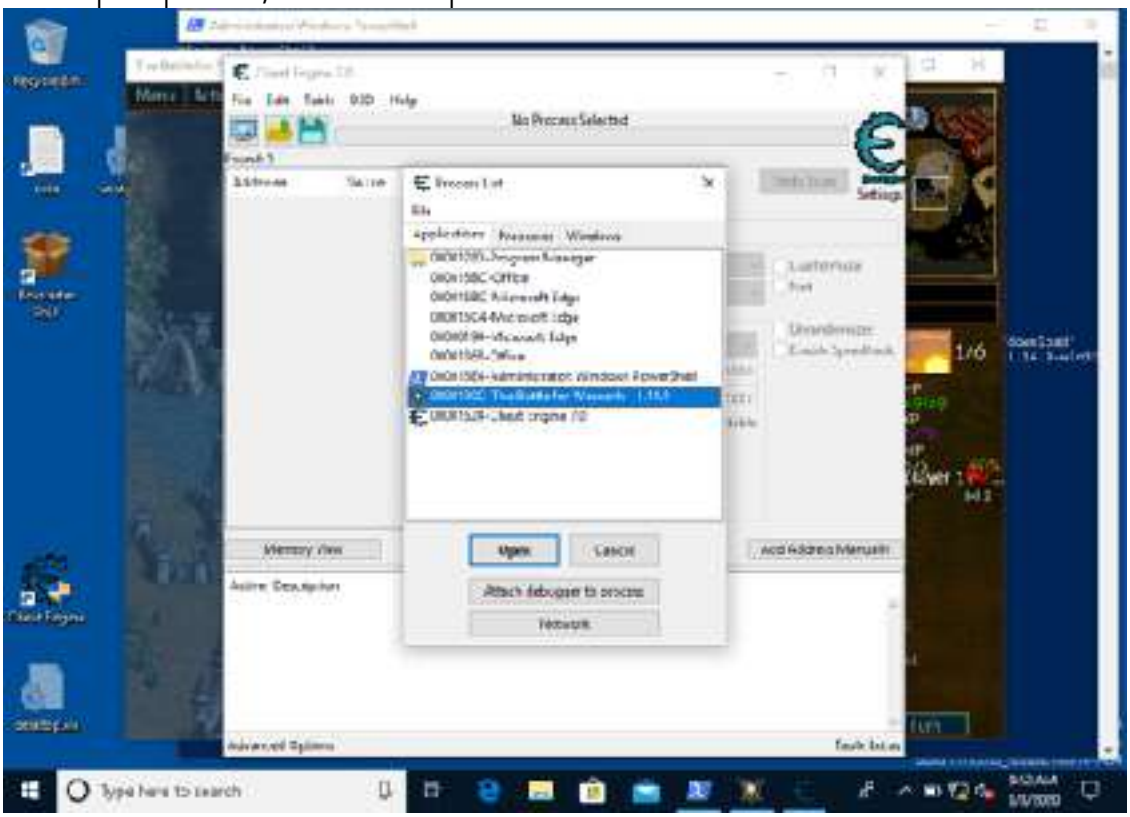
Para completar com sucesso nosso ataque de hacker, precisamos encontrar o local de armazenamento dessa variável na memória e alterar seu valor. Como estamos processando uma variável, usaremos um scanner de memória para encontrá-la.

1.5.4 Posicionamento

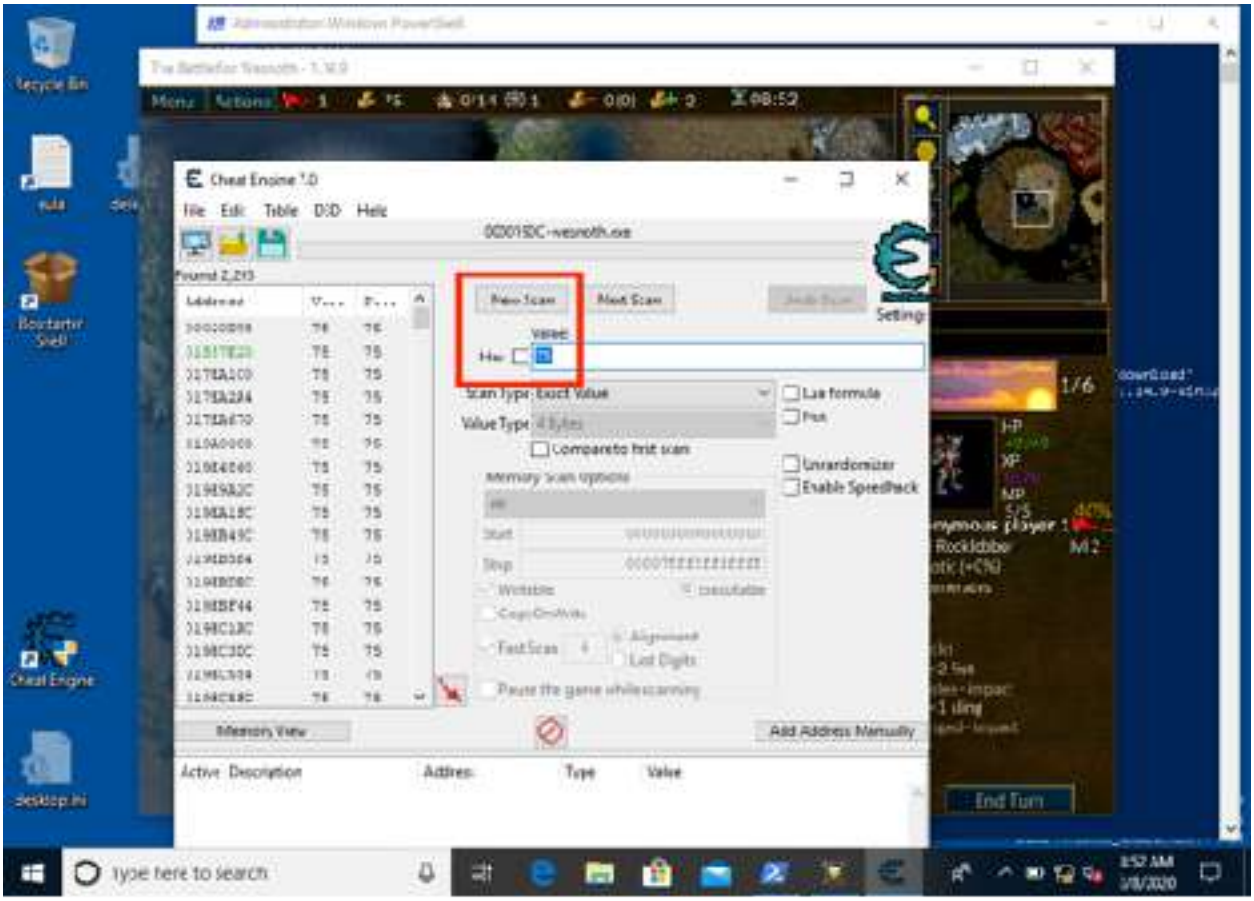
Nosso próximo passo é encontrar memória que preserve o valor do nosso ouro. Começaremos ativando o mecanismo de fraude e adicionando-o ao Wesnoth. Nesta ferramenta, clique no ícone no canto superior esquerdo para parecer um computador com uma lupa.



Na janela que aparece, selecione o processo Wesnoth.

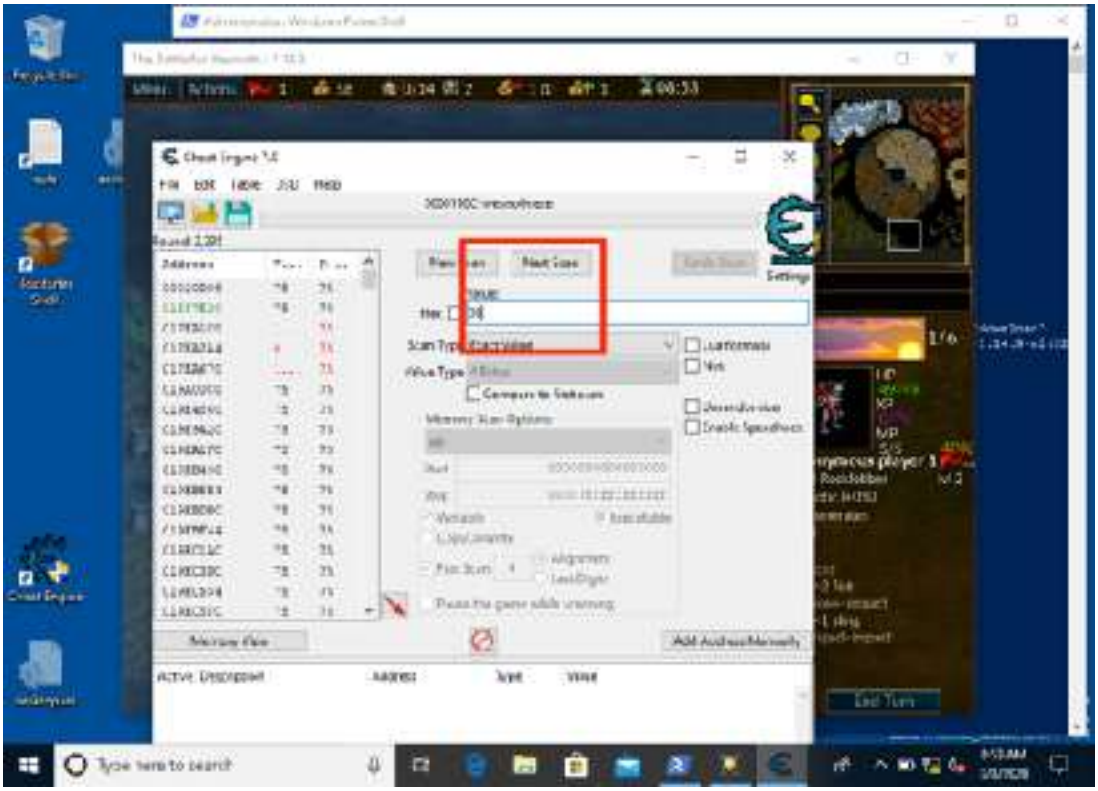


Um scanner de memória permite que você escaneie valores na memória do jogo. No jogo de exemplo, o jogador tem 75 moedas de ouro, então é isso que estamos procurando. Coloque seu valor de ouro na caixa de valores e selecione Nova Scaneamento. Milhares de resultados retornarão.

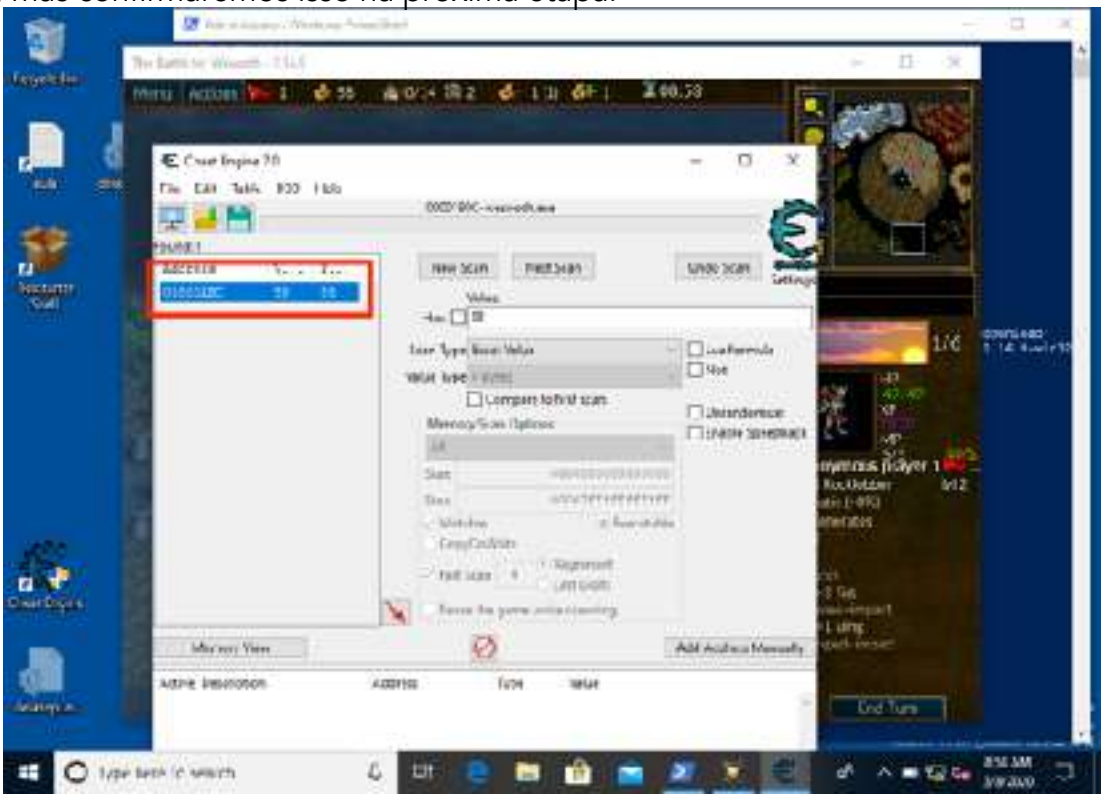


Uma varredura inicial em qualquer scanner de memória retorna milhares de resultados. Isso ocorre porque os jogos são complexos e milhares de memórias são tão valiosas quanto nosso ouro. Essas outras partes podem ser variáveis, como temporizadores, ouro do oponente ou saúde do personagem. Nosso objetivo ao usar scanners de memória é filtrar esses resultados em um ou dois valores, que então podemos testá-los manualmente . Para isso, modificaremos nosso valor de ouro no jogo e, em seguida, usaremos o botão de próxima digitalização para realizar outra digitalização. A próxima operação de varredura retornará apenas o resultado que era anteriormente inicial, neste caso, 75.

Recrute uma força no jogo e veja seu novo preço de ouro. Coloque esse novo valor na caixa "Valores" e selecione "Próxima digitalização". No exemplo a seguir, recrutamos uma unidade para 17 moedas de ouro, deixando 58 moedas de ouro restantes.

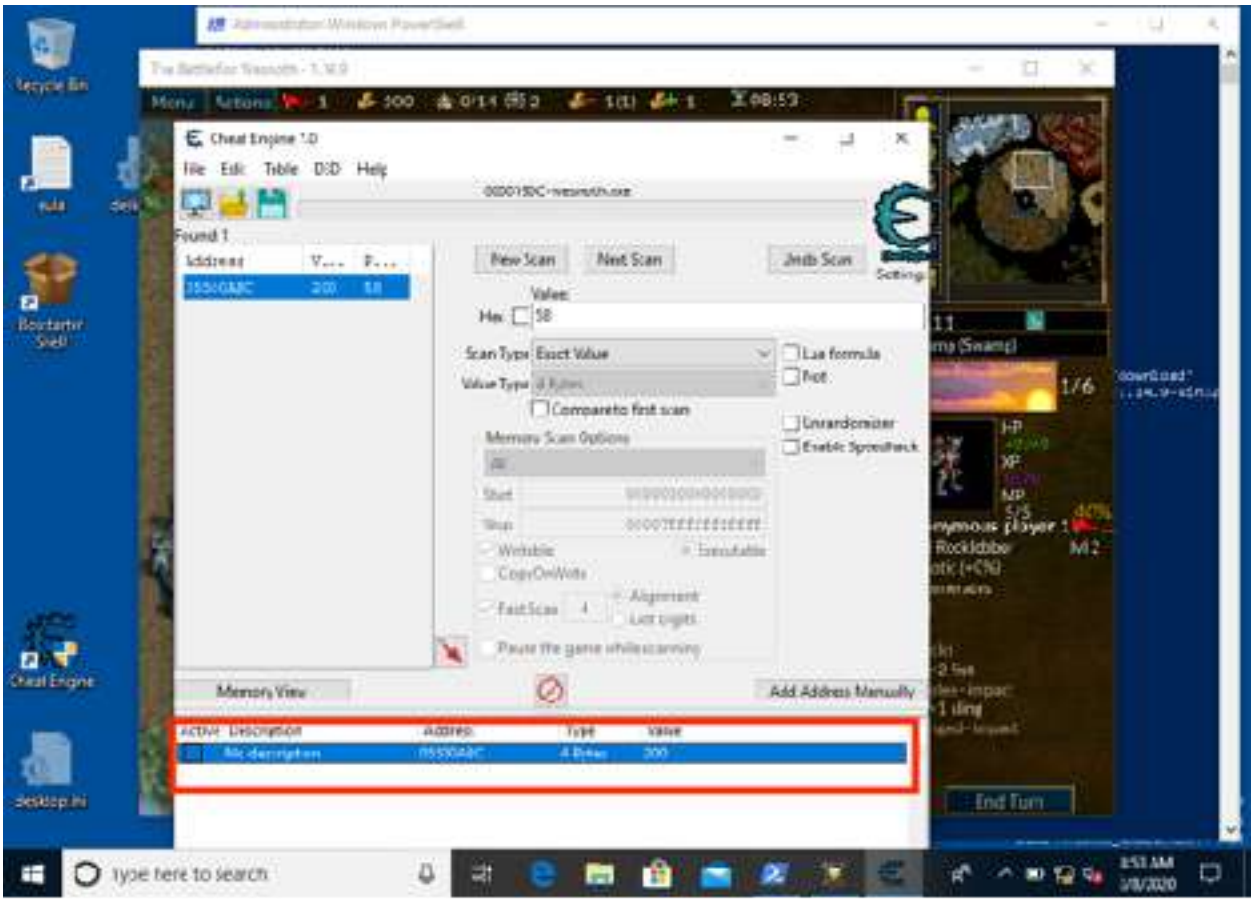


O resultado da segunda varredura tem apenas um valor. Isso provavelmente é o nosso ouro, mas confirmaremos isso na próxima etapa.



1.5.5 Alteração

No mecanismo de fraude, clique duas vezes na posição da memória para movê-la para a caixa na parte inferior da tela. Este bloco inferior permite que você edite valores armazenados na memória. Clique duas vezes em seus resultados para colocá-los na caixa inferior. Em seguida, clique duas vezes no valor e uma caixa aparecerá pedindo que você insira um novo valor. Insira algo grande nele, como 200.



Após a mudança de valor, retorne ao jogo. Você deve ver seu ouro atualizando para 200. Recrutar um grande número de empresas para confirmar que sua mudança foi bem-sucedida.

The Battle for Wesnoth - 1.14.9
Menu
Actions
End Turn
Type here to search

Parte 2
Depuração e Ré

2.1 Depuração Fundamentos

2.1.1 Objetivo

No capítulo anterior, prejudicamos nosso ouro modificando uma variável na memória. Características de modificação de memória como esta são poderosas, mas também têm limitações . Hackers mais complexos geralmente exigem que você modifique o código do jogo. Por exemplo , imagine se quiséssemos criar um hacker que nos permitiríamos recrutar uma unidade sem ouro. Uma maneira é monitorar constantemente nosso ouro e adicioná-lo manualmente à medida que recrutamos uma unidade. Isso também exige que você procure constantemente quaisquer novas unidades adicionadas ao jogo e ao custo dessa unidade. Uma abordagem mais simples é modificar o código do jogo, o que nunca reduzirá o dinheiro do jogador ao recrutar.

Verificar o código enquanto o jogo está em execução é chamado de depuração. Entender e modificar o código para fazer o que você quer é chamado de reversão. Você não precisa ajustar um jogo para revertê-lo, mas isso é muito útil, se possível.

2.1.2 Ferramentas envolvidas

Para depurar o jogo, você precisa usar uma ferramenta chamada depurador. O primeiro passo na depuração é "anexar" o depurador ao jogo que você deseja depurar . Uma vez conectado, você pode visualizar o código do jogo na memória. Você também pode pausar a execução do jogo, alterar o código do jogo e modificar o registro. Veremos exemplos dessas ações em capítulos posteriores.

O depurador pode causar efeitos colaterais inesperados. Por exemplo, se você alterar erroneamente o código do jogo, o jogo pode travar. Dependendo do jogo, isso pode congelar a exibição do seu computador. Essa é outra razão pela qual você sempre separará seu hacker da sua máquina pessoal.

Existem muitos depuradores, mas alguns famosos incluem IDA e gdb. Outros depuradores, como WinDbg e OllyDbg, são frequentemente mencionados, mas não são mais mantidos. Nesta série, usaremos um depurador de código aberto chamado x64dbg. Assim como todo mundo

Ferramentas: Entender os fundamentos é mais importante do que entender as ferramentas. Aplicamos a mesma abordagem que aprendemos ao usar o x64dbg a qualquer depurador.

2.1.3 Desmontagem e depuração

Depois que o depurador for adicionado ao jogo, ele exibirá o código do jogo. Entretanto, este não é o código do jogo original. Como discutimos no Capítulo 1, os jogos geralmente são programados em linguagens de alto nível, como C++. Entretanto , os arquivos executáveis que são executados em nossos computadores contêm apenas o código de operação que a CPU precisa executar. Essa falta de código bruto dificulta a reversão. Os jogos geralmente contêm milhares desses códigos de operação.

Ao montar um programa, cada linha de código de montagem é convertida em um código de operação. Desmontagem é o processo de conversão de códigos de operação de volta para montagem. Normalmente, desmontagem e depuração podem ser usados de forma intercambiável, especialmente em jogos de reversão No entanto, eles podem ser feitos separadamente. O programa pode ser desmontado sem depuração. Isso é chamado de análise estática e geralmente é feito quando o malware é reverso. O programa também pode ser depurado sem desmontar. Um exemplo comum é depurar o programa que você escreveu. Neste caso, você tem o código original e não é necessário desmontagem.

O código avançado original pode ser recriado da seção de desmontagem. Isso é o que se chama de descompilação. Entretanto, a descompilação sempre representa o código que está sendo executado, enquanto a descompilação não. Os compiladores são frequentemente forçados a adivinhar a estrutura original do código. Embora essas ferramentas sejam úteis, elas também podem levá-lo no caminho errado. Neste livro, não abordamos a descompilação.

2.1.4 Montagem

Ao depurar e reverter um jogo, você lidará principalmente com a montagem. A linguagem de assembleia é semelhante à primeira linguagem que apresentamos no Capítulo 1.1. Cada instrução na compilação faz uma coisa, como adição ou subtração. Não há necessidade de conhecer cada comando de montagem para reverter um jogo.

Embora isso possa parecer assustador, qualquer código de montagem pode ser entendido por instrução de cada vez. Durante a depuração, isso é chamado de desmontagem passo a passo. Normalmente, há muitos comandos no jogo que não são importantes para entender ao dar ré. Por exemplo, ao somar números com valores decimais, a CPU deve executar várias instruções. Se somente fazemos

O resultado dessa adição é que podemos pular muitas dessas instruções. Entender quais instruções podem ser ignoradas requer experiência.

2.2 Ré Fundamentos

2.2.1 Contexto

Ao adicionar um depurador ao jogo pela primeira vez, o processo de reversão do jogo pode ser confuso. A melhor maneira de começar a reverter um jogo é descobrir o que você quer ver e depois descobrir onde ele está. Depois de estabelecer esse contexto, você só pode completar passo a passo as instruções que são realmente importantes para você.

Existem muitas maneiras de criar contextos. Em alguns casos, você pode querer pesquisar o texto exibido quando o jogo executa uma operação. Qualquer local onde este texto é carregado deve, em última análise, estar relacionado à ação na qual você está interessado. Em outros casos, você pode usar o endereço de memória encontrado no editor de memória para encontrar o código do seu interesse. Independentemente da abordagem que você adotar, os pontos de interrupção serão usados.

2.2.2 Ponto de Interrupção

O ponto de interrupção permite que o depurador pague a execução do jogo em um comando específico. Após a pausa do jogo, você pode completar passo a passo um único comando e visualizar a memória do jogo. Você pode definir pontos de interrupção em qualquer tipo de memória. Isso inclui memória encontrada usando um scanner de memória.

Os pontos de interrupção podem ser definidos para acionamento incondicional e condicional. Pontos de interrupção condicionais só são acionados quando suas condições são atendidas. Essas condições podem ser que o registro tenha um valor ou uma mudança na memória (onde o ponto de interrupção é definido). Quando um ponto de interrupção é acionado, isso também é chamado de pop-up.

2.2.3 Pontos de interrupção de memória

A melhor maneira de ilustrar como usar pontos de interrupção é por meio de um exemplo. Nesta seção, exploraremos como usar pontos de interrupção de memória para criar contextos.

Voltando ao Capítulo 1.5, encontramos a posição da memória do ouro. Podemos usar esse local de memória para encontrar a lógica do jogo responsável por reduzir nosso ouro. Fizemos isso definindo um ponto de interrupção condicional em nosso local de memória dourada e depois entrando no jogo para recrutar uma unidade. Quando recrutamos unidades, pontos de interrupção aparecem e a execução parará na função responsável por reduzir o ouro do jogador. Essa função pode ser mostrada abaixo, onde a linha azul representa o local onde pausamos.

```
mov ebx, dword ptr ds: [0x05500ABC]
mov ebx, dword ptr ds: [0x12345678]
sub eax, ebx
mov dword ptr ds: [0x05500ABC], eax
Mi é, ebx
```

A primeira instrução move o valor armazenado em 0x05500ABC para o registro eax. Este valor é a posição dourada que encontramos no capítulo anterior. A próxima instrução move o valor assumido do custo unitário para o registro ebx. O jogo então deduz o custo unitário do nosso valor de ouro. Nossa posição de pausa é responsável por mover novos valores de ouro de volta para a posição de memória onde nossos valores de ouro são armazenados.

Você pode notar que o jogo não pausa durante as operações de subtração. Isso ocorre porque esta operação apenas modifica o valor no registro, não o valor real da memória onde definimos o ponto de interrupção. O ponto de interrupção sempre será pausado no comando imediatamente após a memória ser afetada.

2.2.4 Ponto de interrupção do código

Às vezes, pode ser difícil ou impossível encontrar o valor da memória para definir o ponto de interrupção. Nesses casos, você pode definir pontos de interrupção em um pedaço de código. Um exemplo comum é definir um ponto de interrupção em uma referência de texto e usá-lo para encontrar funções de nível superior que nos interessam.

Considere um jogo para o qual queremos escrever wallhack. O loop principal do jogo pode ser semelhante a:

```
void main_loop(){
draw_players();
DRAW_Wls();
...
```

```
}
```

A função draw_wall do jogo pode ser semelhante a:

```
void draw_wall(){
bool if(subcedido==falso){

print_error();}

}
```

Por fim, a função PRINT_EROR pode ser semelhante a:

```
void print_error(){
print_to_log ("textura da parede não pode ser
encontrada");}
```

Uma maneira de escrever o wallhack é remover a função draw_wall do jogo. Como não há variáveis que possam ser usadas como pontos de interrupção de memória, usaremos pontos de interrupção de código.

O depurador permite que você visualize todo o texto no jogo e todos os locais onde ele é usado. Por exemplo, usando o depurador, podemos encontrar texturas de parede e locais de referência que não podem ser encontrados. Pode parecer:

```
mov eax, dword ptr ds: [0x23456789]
eax push
Chamar print_to_log
…
```

Este código é responsável por carregar strings no registro e então chamar a função print_to_log. Ao definir um ponto de interrupção neste código e encontrar uma textura perdida no jogo, nosso ponto de interrupção aparecerá. Podemos então continuar executando o código em uma única etapa até que ele nos devolva à função que chamou o código. Isso é chamado de função de saída. Após sair, entraremos na função draw_wall e poderemos excluí-la.

2.2.5 Instruções NOP

nop (código de operação 0x90) significa nenhuma operação. Quando esse comando é encontrado, a CPU não fará nada e continuará com o próximo comando . Esse comportamento pode ser usado para modificar a lógica do jogo.

Por exemplo, na Seção 2.3.3 acima, encontramos a seção de código responsável pela subtração de ouro. O código é o seguinte:

```
mov ebx, dword ptr ds: [0x05500ABC]
mov ebx, dword ptr ds: [0x12345678]
sub eax, ebx
mov dword ptr ds: [0x05500ABC], eax
```

Ao substituir suboperações por nop, o jogo não subtrairá mais nossas moedas de ouro.

2.3 Mudança de jogo Código

2.3.1 Objetivos

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

2.3.2 Identificação

Nosso objetivo neste capítulo é mudar o código do Wesnoth para que os recrutadores não reduzam nosso ouro.

2.3.3 Compreensão

Para modificar o código do jogo, precisamos usar um depurador. Para encontrar o código do jogo a ser modificado, definimos um ponto de interrupção em nosso endereço dourado e então recrutamos uma unidade. Nosso depurador exibirá um código responsável por reduzir nosso ouro. Então podemos omitir subinstruções.

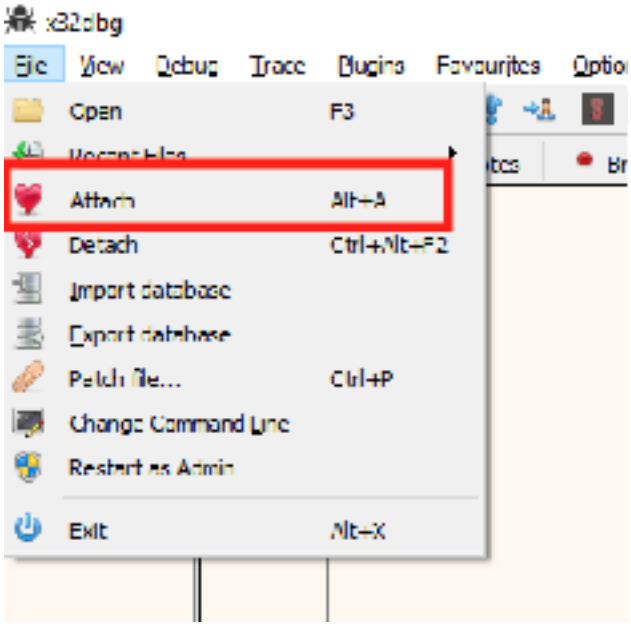
2.3.4 Posicionamento Ouro

Nosso primeiro passo foi abrir o Wesnoth e criar um jogo local. Você pode então seguir os passos do Capítulo 1.5 para encontrar seu endereço dourado. Devido a um processo chamado alocação dinâmica de memória (ou DMA), ele estará localizado em um endereço diferente do anterior. Introduziremos o DMA em capítulos posteriores. Assim que você encontrar seu novo endereço dourado, desligue o mecanismo de fraude, mas mantenha Westminster aberto. Neste capítulo, usaremos

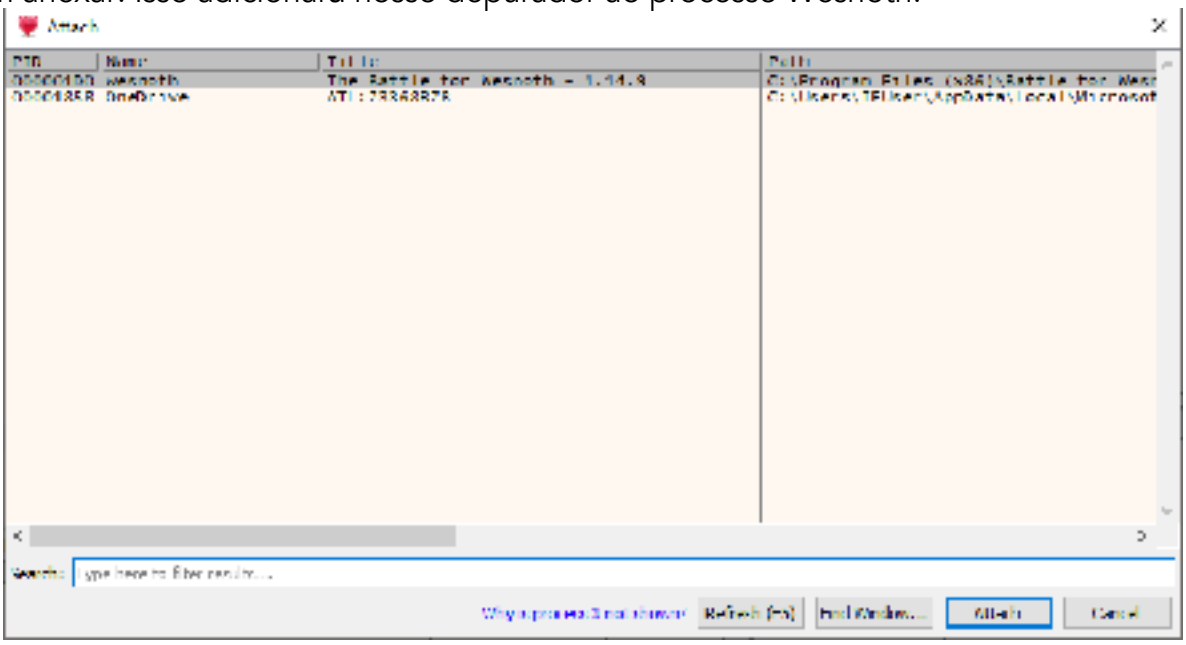
0x051D875C é nosso endereço privilegiado.

2.3.5 Depurador adicional

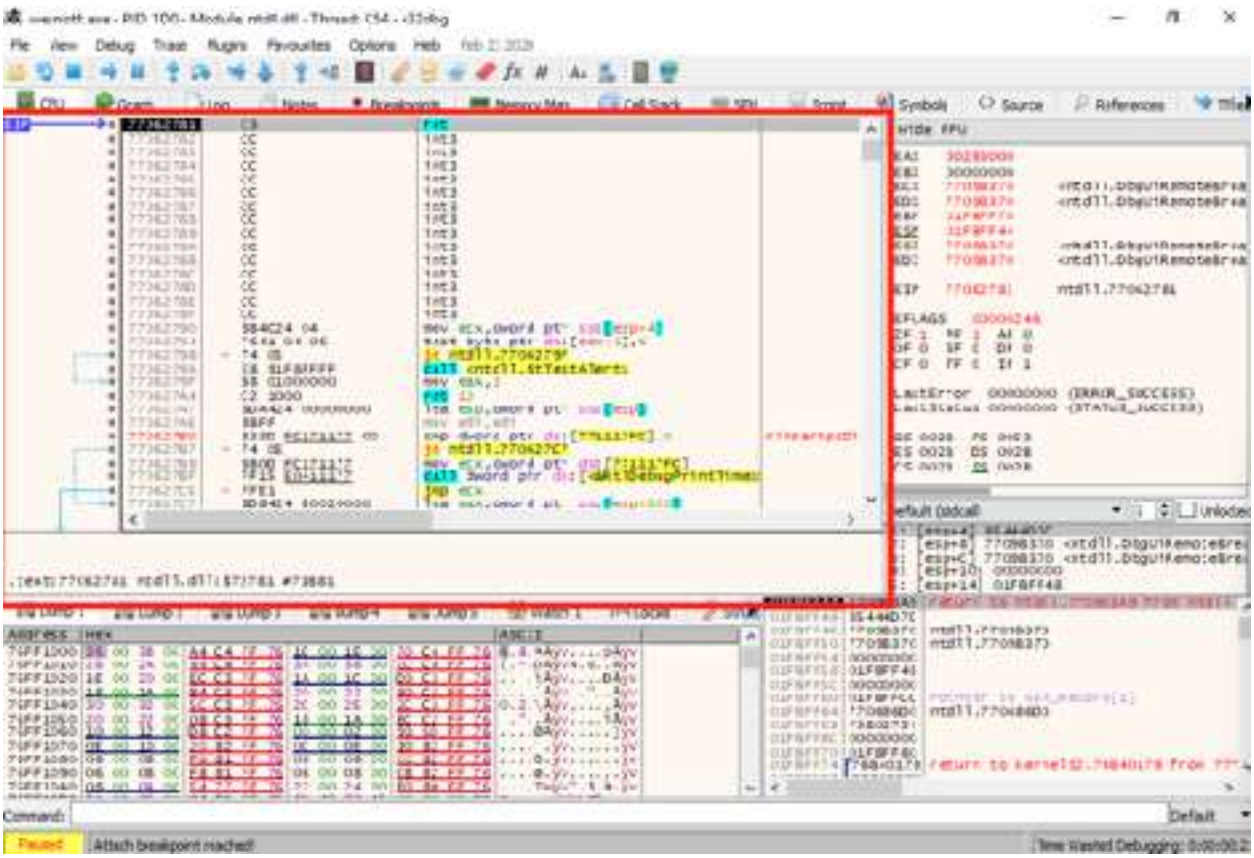
Em seguida, inicie a versão de 32 bits do x64dbg. Ele pode ser encontrado em C:\ ProgramData \Chocolatay \lib\x64dbg. portables \tools\release \x32\x32dbg. exe. Após iniciar, abra o menu de arquivos e selecione Adicionar.



Na caixa de diálogo de anexação aberta, selecione o processo Wesnoth e clique em anexar. Isso adicionará nosso depurador ao processo Wesnoth.



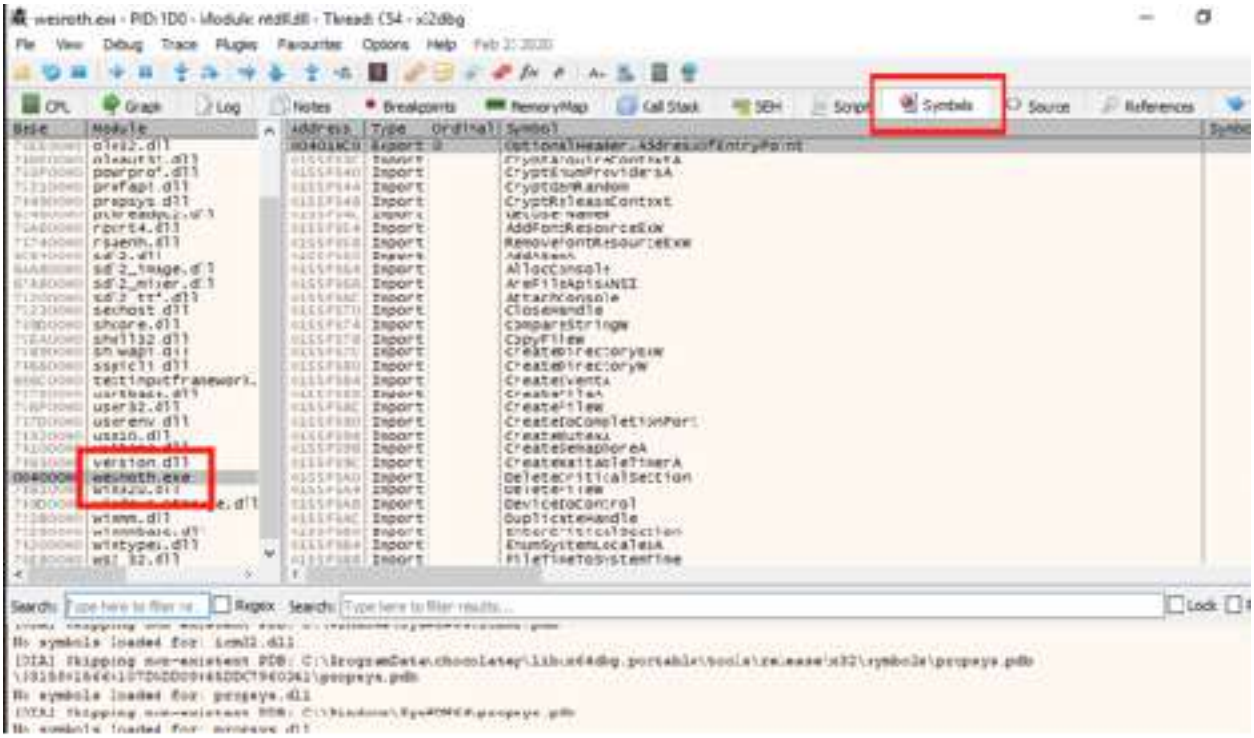
Após a conexão, o x64dbg exibirá um módulo chamado ntdll.dll e exibirá uma grande quantidade de informações.



Antes de nos aprofundarmos no reverso, vamos dar uma olhada rápida nos principais componentes de qualquer depurador. A seção destacada contém o código que está sendo executado. A seção de dump diretamente abaixo exibe a memória do aplicativo em formato hexadecimal (hex) e ASCII. No lado direito da seção de código há uma lista de todos os registros e seus valores. Abaixo do registro está a pilha de aplicativos.

Neste capítulo, usaremos apenas a parte de código e a parte de dumping. Entender todas essas partes é essencial para hackers mais complexos. Diferentes depuradores sempre conterão todas essas informações, mas geralmente as organizam de maneiras diferentes.

Atualmente, estamos analisando o módulo ntdll.dll. Este não é o nosso objetivo, mas é um módulo universal carregado em todos os arquivos executáveis do Windows. Para visualizar o código do jogo, precisamos navegar para a aba de símbolos e clicar duas vezes em wesnoth.exe.

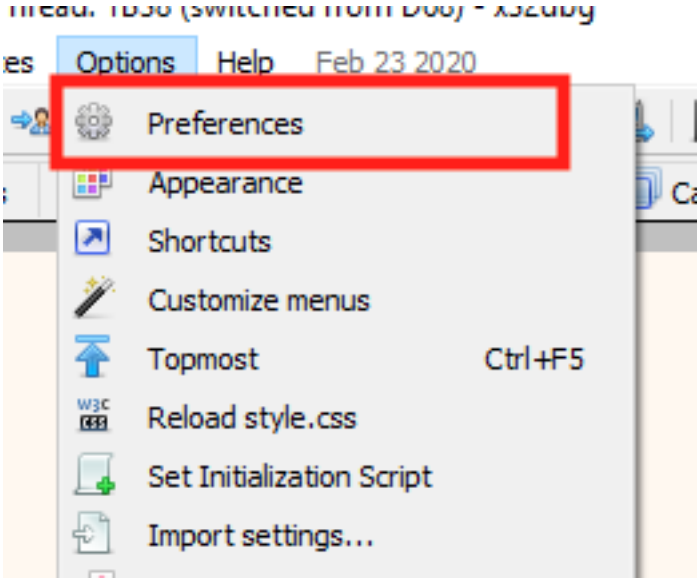


Isso mudará nossa visão para o código do jogo e o espaço de memória.

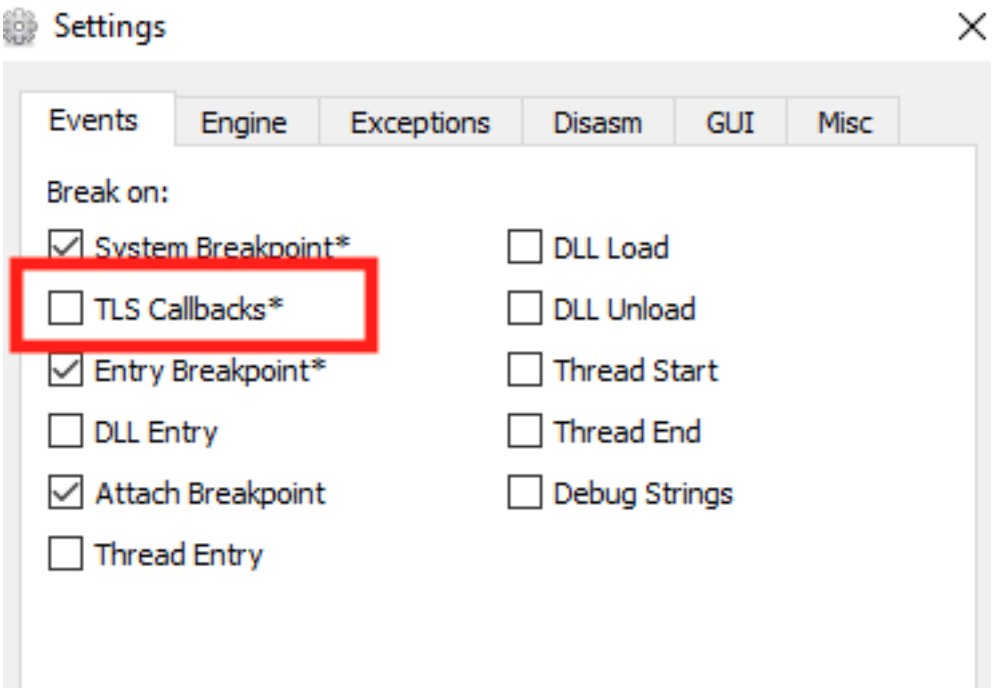
2.3.6 Configurar o depurador

Algumas configurações padrão no x64dbg fazem com que ele pop-up quando não queremos que ele pop-up. Para facilitar a marcha ré neste capítulo e capítulos subsequentes, desativaremos essas configurações.

No menu superior, selecione a opção -> preferência.



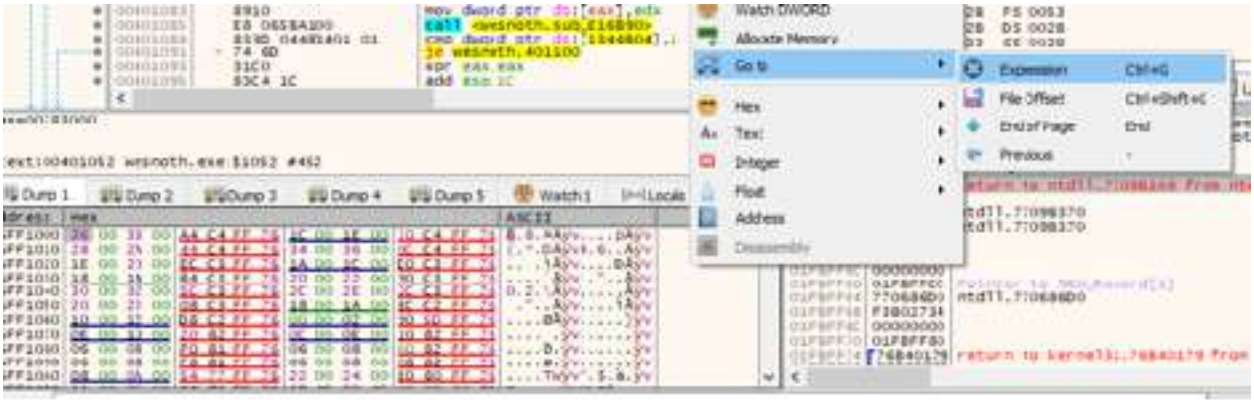
No modo aberto, desselecione a opção de retorno do TLS e selecione Salvar. Isso desabilitará o x64dbg de pop-up automaticamente ao receber um retorno do TLS.



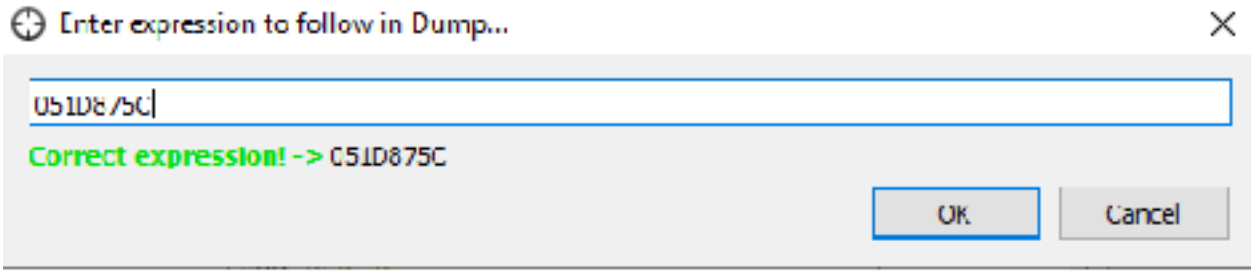
2.3.7 Defina pontos de interrupção

Em seguida, definimos um ponto de interrupção em nossa posição dourada. Depois que definimos esse ponto de interrupção, entraremos em Vistanos e recrutaremos uma unidade. Isso fará com que o ponto de interrupção apareça e a execução seja suspensa na posição responsável pela dedução do ouro.

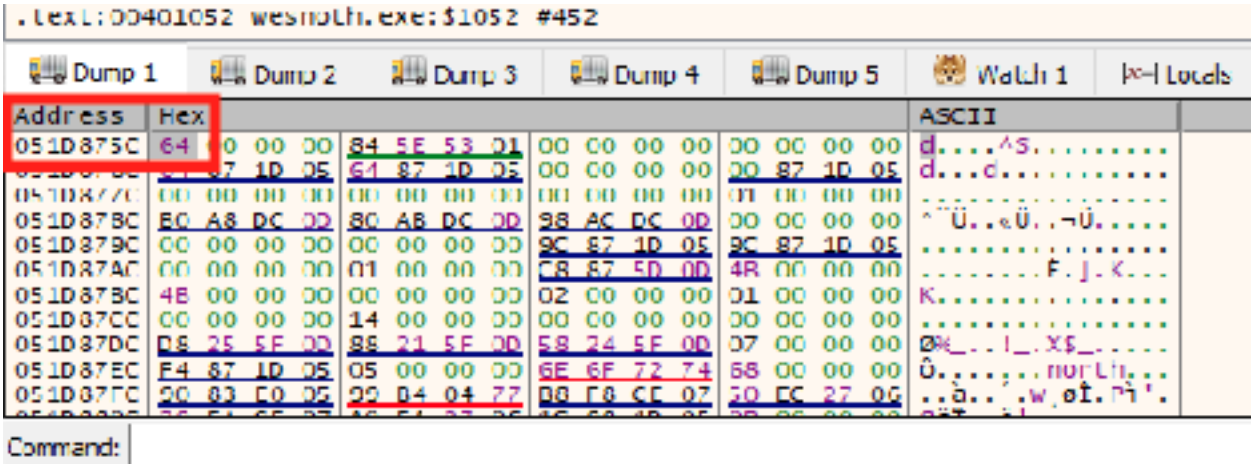
Clique com o botão direito na seção de dumping e selecione ir para -> expressão:



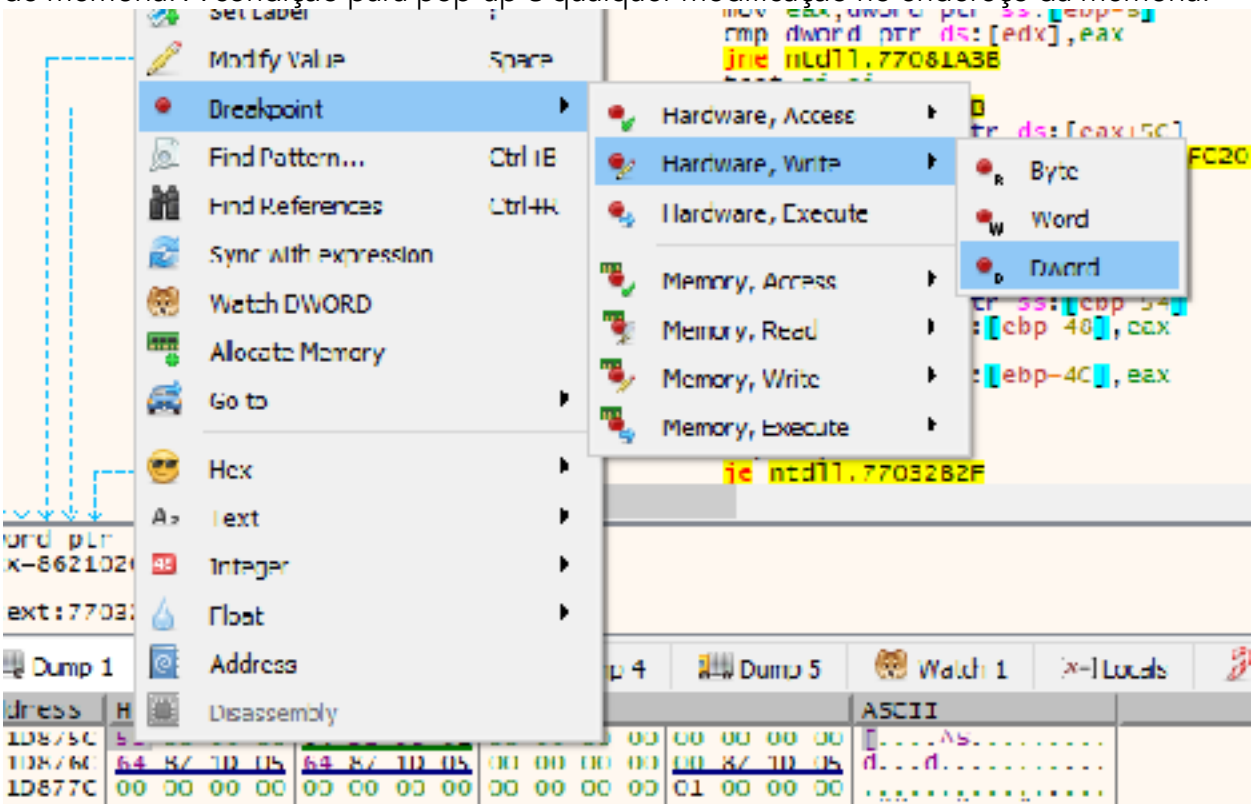
Na caixa de diálogo aberta, insira nosso endereço dourado e clique em OK.



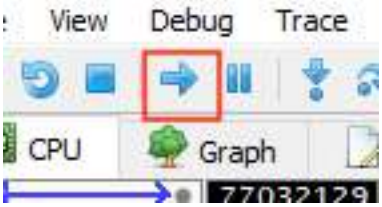
O dump exibirá o endereço que acabamos de inserir. Os dados exibidos estão em formato hexadecimal. No jogo de alvo, os jogadores têm 100 moedas de ouro. 100 no formato hexadecimal é 0x64. Este é o valor exibido no dump.



Clique com o botão direito no valor e selecione o ponto de interrupção ->Hardware, escreva em ->DWORD. Isso definirá um ponto de interrupção condicional neste endereço de memória. A condição para pop-up é qualquer modificação no endereço da memória.

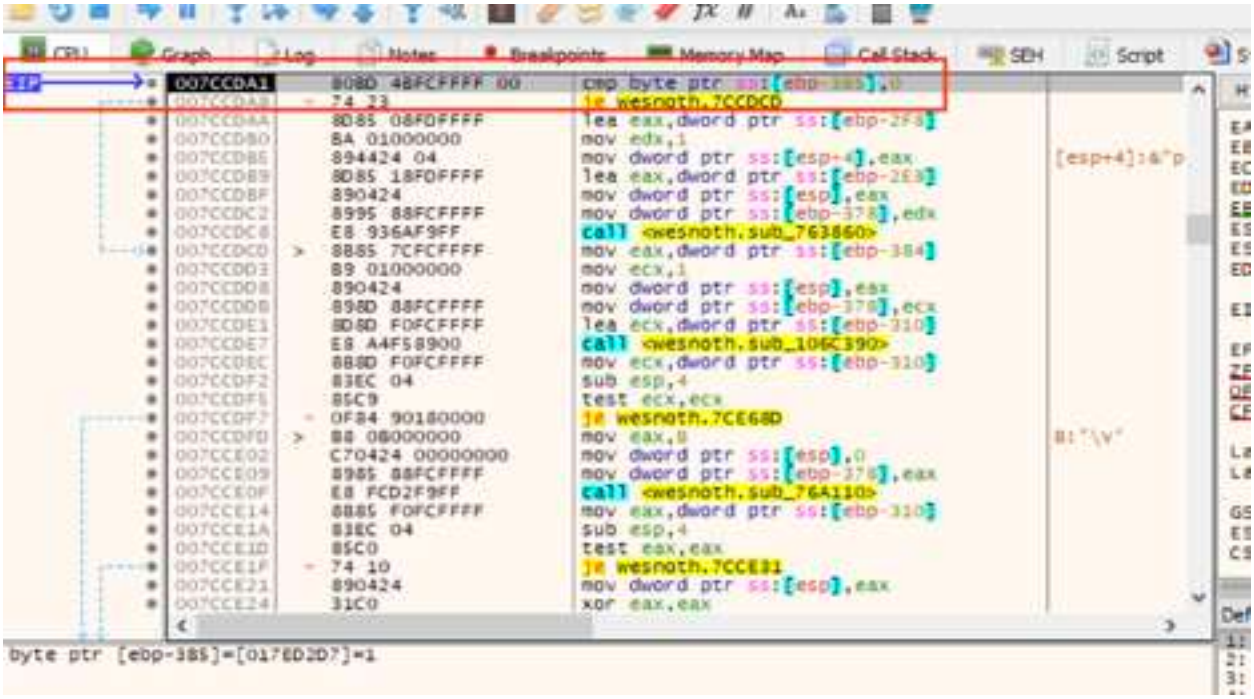


Depois de definir o ponto de interrupção, agora retomaremos a execução do programa. Isso pode ser feito pressionando o botão de reprodução até que a pausa desapareça e o jogo seja retomado. Como o x64dbg cria automaticamente vários pontos de interrupção, você terá que fazer isso várias vezes.



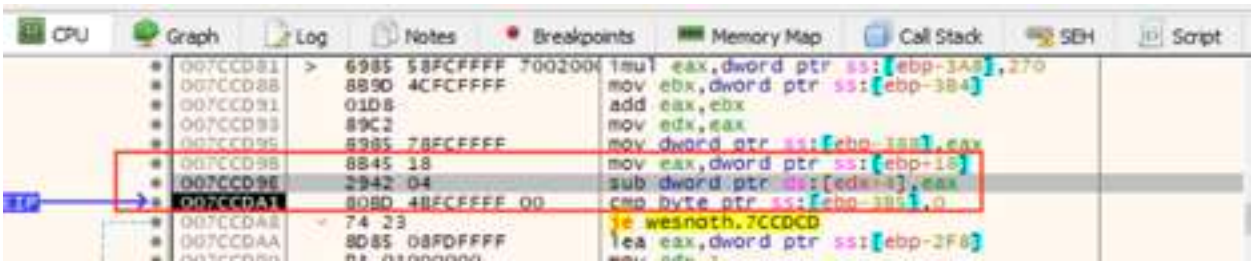
2.3.8 Código de Localização

Assim que o jogo for retomado, retorne a Westnors para recrutar uma unidade. Você notará que o Wesnoth ficará congelado devido ao pop-up e à pausa da execução do nosso ponto de interrupção. Navegue de volta para x64dbg para ver onde ele pop-up.



O EIP destacado representa o local de execução atual no programa. EIP representa um ponteiro de instrução estendida, que é um registro especial usado pelo programa para entender o local de execução atual.

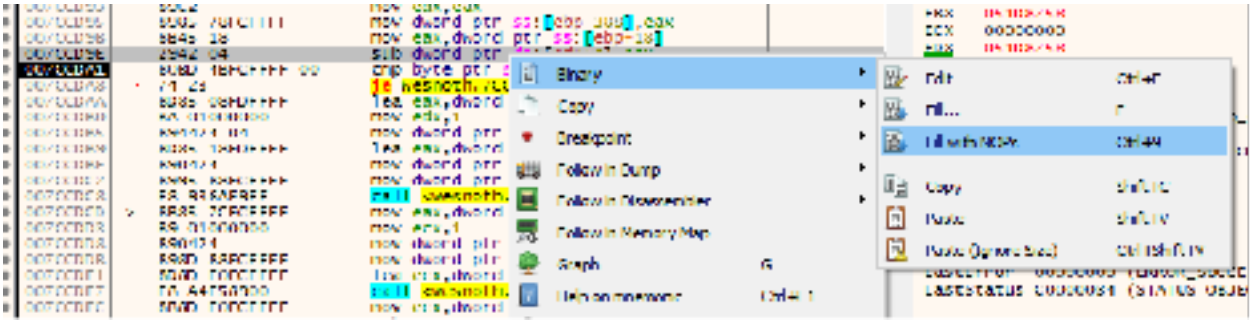
Como sabemos no capítulo anterior, pontos de interrupção condicionais são acionados após operações que afetam a memória. Role para cima na janela de código para ver as instruções anteriores.



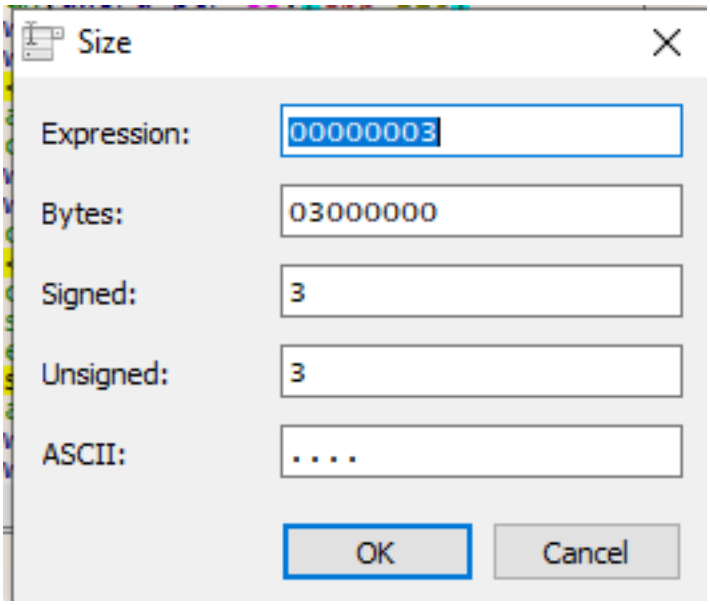
As subinstruções destacadas são responsáveis por modificar nosso valor de ouro. Como lembramos nos capítulos anteriores, sub representa subtracte e é responsável por subtrair dois números. Neste caso, ele subtrai os valores armazenados nas posições de memória em edx+4 e eax. Não há necessidade de saber os detalhes exatos desses valores agora. Tudo o que precisamos saber é que essa operação está afetando nosso ouro de alguma forma.

2.3.9 Alteração

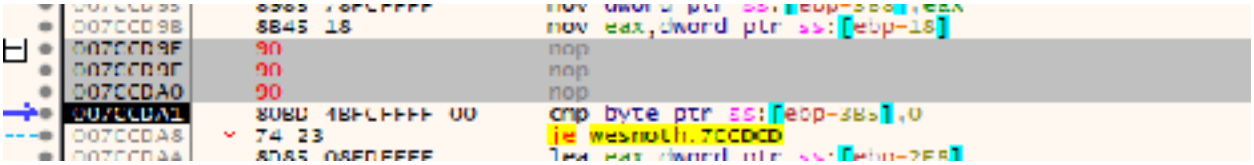
Por fim, alteraremos esse código e completaremos nosso ataque de hacker. Para isso, substituiremos o comando sub por comando nop. Isso substituirá a subtração por uma operação que não faz nada. Dessa forma, nosso ouro não diminuirá mais. Felizmente, o x64 dbg inclui um método integrado para excluir instruções automaticamente. Clique com o botão direito na linha com instruções de subtração e selecione binário -> para preenchê-la com NOP.



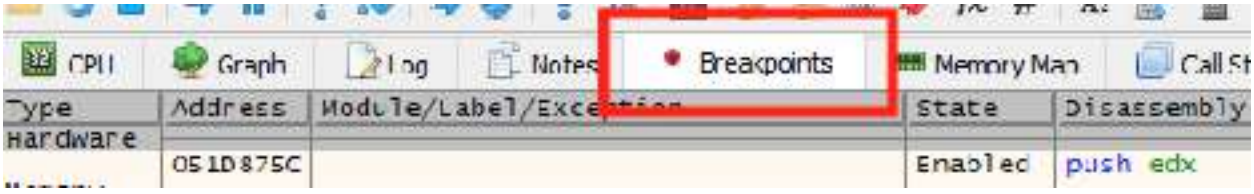
x64dbg preencherá automaticamente o próximo valor para você. Basta selecionar OK na próxima tela.



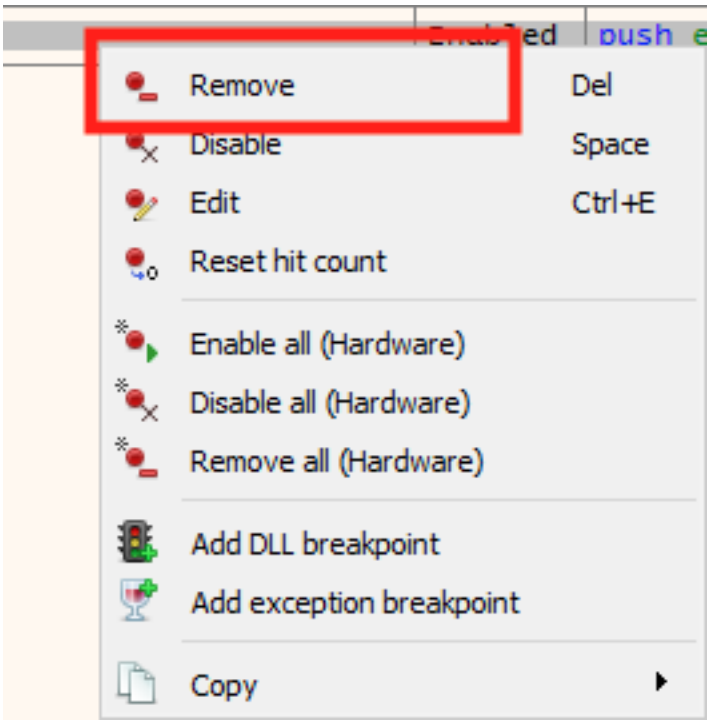
Se for feito corretamente, o código agora deve ser mostrado na figura abaixo. Você notará três instruções nop. Discutiremos os motivos em capítulos posteriores.



Antes de verificar se nossas alterações são válidas, precisamos desativar nosso ponto de interrupção para que ele não apareça novamente. Para isso, vá primeiro para a aba de ponto de interrupção. Esta aba contém uma lista de todos os pontos de interrupção que definimos no aplicativo.



Clique com o botão direito no ponto de interrupção definido e selecione Excluir.



Depois de remover o ponto de interrupção e alterar o código, agora podemos retornar ao Wesnoth e observar nossas mudanças. Recrutar algumas unidades e observe que seu ouro não cai mais.

2.4 Código de ré

2.4.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

2.4.2 Identificação

Para recrutar na Wesnoth, você pode clicar com o botão direito em um adesivo e selecionar Recrutamento. Na Wesnoth, você só pode recrutar unidades em certos ladrilhos. O objetivo do nosso capítulo é mudar esse comportamento para que possamos recrutar unidades em qualquer lugar do mapa.

2.4.3 Compreensão

Este hacker exige que usemos um depurador para modificar o código do jogo. Para realizar esse tipo de ataque de hacker, primeiro precisamos encontrar o código que é executado ao clicar com o botão direito em um bloco e selecionar uma opção. O código original do jogo pode ser o seguinte:

```
Interruptor (Option_Select){
Caso "Descrição do Terreno": show_terrain_
description (localização); descanso;

Caso "recruit": recruit_unit (
localização); descanso;

Caso...}
```

A declaração de troca permite que você execute diferentes ramos com base no estado de uma variável. Queríamos modificar esta frase para que clicar na descrição do terreno chamasse o código da unidade de recrutamento.

2.4.4 Bolhas

No capítulo anterior, encontramos o código responsável pela redução de dinheiro ao recrutar unidades. Usaremos este código para estourar até o local do código do menu com clique direito. Para ilustrar o conceito de bolhas, imagine o código usado para unidades de recrutamento em Wesnoth é o seguinte:

```
Função handle_context_menu(){
...
Caso "recruit": recruit_unit (
localização); descanso;

...
}
...
Função Recruit_Unite{
...
Check _localização (localização);
find_unit_in_unit_list();
...
```

```
}
...
Função Find_Unit_In_Unit_Unit_LIST(){
...
get_unit();
get_unit_cost();
Subtracta_Unit_Cost();
...
}
...
Função subtracta_Unit_Cost(){
...
Check_Player_Good();
Subtracta_Good();
...
}
...
Função subtracta_GOLD(){
player_money=player_money-cost_of_unit;
}
```

Uma boa maneira de visualizar as interações entre todas essas funções é usando cadeias de funções. A cadeia de funções neste exemplo é a seguinte:

```
handle_context_menu()->recruit_unit()->Find_Unit_In_Unit_Unit_Unit_
Unit_Cost()->subtract_unit_cost()->Subtract_Good()
```

O código que encontramos no capítulo anterior está na função subtract_Good. Começando com este código, eventualmente encontraremos a função handle_context _menu.

Para fazer bolhas em nosso depurador, aproveitaremos dois recursos: execução até retornar e passos. Executar até a função de retorno Executar instruções até chegar à declaração de retorno. A função de passo executa uma linha de código. Ao contrário do recurso Step Into, se a instrução em execução for uma chamada, o Step over não entrará na função. Explicaremos isso em detalhes mais tarde, mas primeiro precisamos explicar como as funções são convertidas em assembleias.

2.4.5 Chamadas e retornos

As instruções de chamada são usadas para chamar funções em uma assembleia. No final da função chamada, a instrução retn (Retorno) é **usada para devolver o código que chamou a função. Por exemplo, o código a seguir usa uma chamada para adicionar o registro eax em 1:**

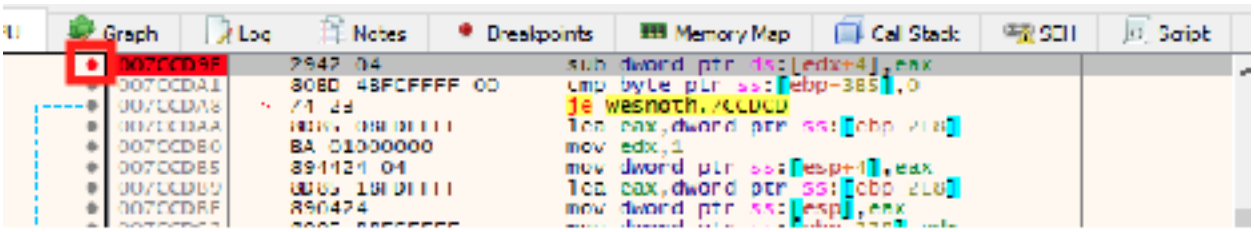
```
Anfitrião:
MiEAX, 0
Aumento de chamada _
eax mov ebx, eax

increase_eax:
adicione eax, 1
mov ecx, eax retn
```

Suponha que definimos um ponto de interrupção no comando add eax, 1. Assim que ele aparecer, usar o recurso Excute till return fará com que o depurador continue executando o código até que a primeira instrução de return seja alcançada. Uma vez no comando retn, o recurso de passo executará o comando retn e chegará ao comando mov ebx e eax. Este é um bom exemplo de bolhas para funções mais altas.

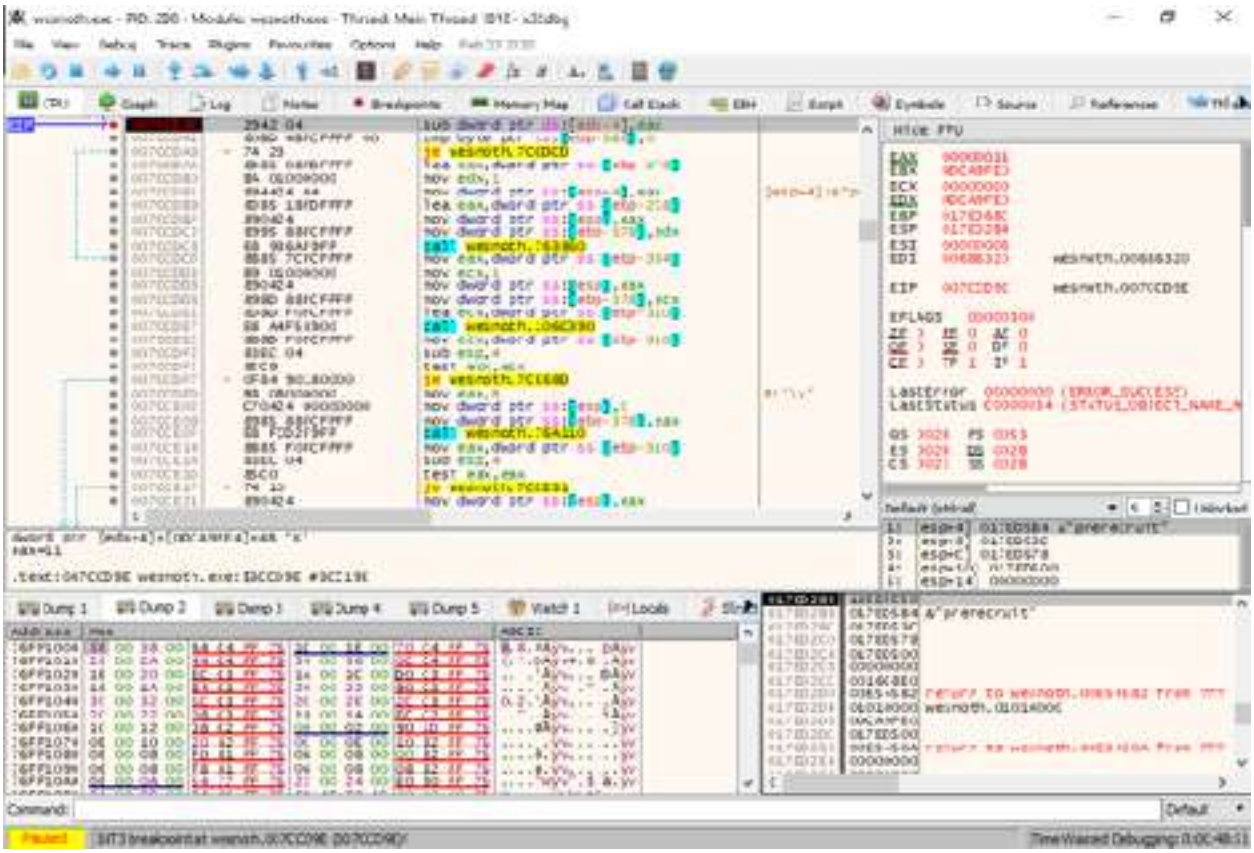
Para entender passo e passo, suponha que definimos um ponto de interrupção ao chamar a instrução increase_eax. Entrar nesta instrução fará com que nosso depurador vá para a primeira linha da função (adicione eax, 1) e espere lá. Pular essa função fará com que nosso depurador continue executando até chegar à instrução mov ebx, eax. Ao lidar com grandes quantidades de código de baixo nível, geralmente é conveniente pular funções para evitar perder tempo.

2.4.6 Menu de Posicionamento

Ao contrário das variáveis, a posição do código no jogo geralmente não muda. Portanto , podemos iniciar a reversão usando a mesma posição encontrada no capítulo anterior. Após adicionar x64dbg ao Wesnoth, navegue para o local que encontramos no capítulo anterior (0x007cd9e) e clique no ponto à **esquerda da instrução para definir o ponto de interrupção. O endereço ficará vermelho, indicando que um ponto de interrupção foi definido. Sempre que esta instrução** é **executada, o ponto de interrupção aparece.**



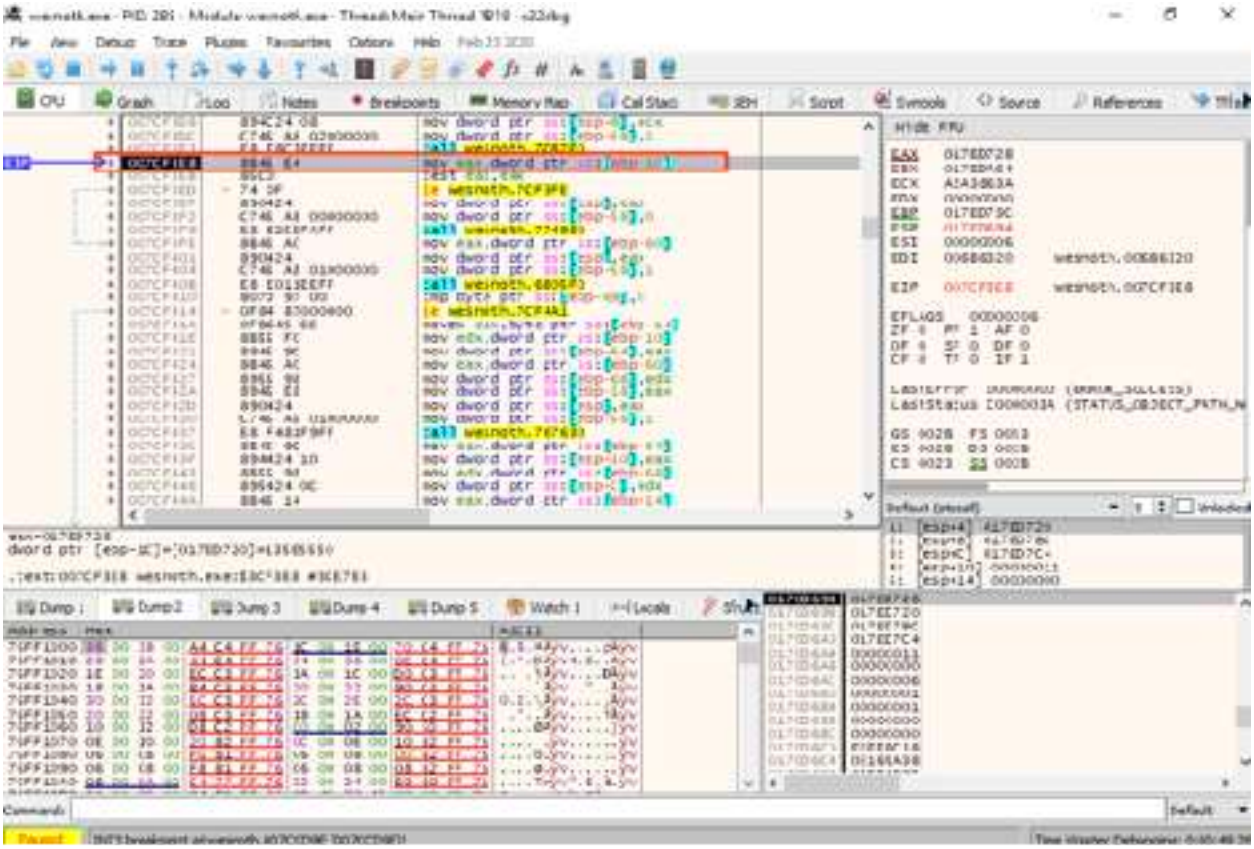
Em seguida, retorne a Vistanos para recrutar uma força. Ao fazer isso, o depurador aparecerá na mesma posição que vimos no capítulo anterior.



Clique no botão "Executar até retornar" uma vez para executar até a primeira instrução de retn.

Após abrir, clique no botão "Pular" para ir para o código de chamada.

Você deve ser enviado para os seguintes locais:



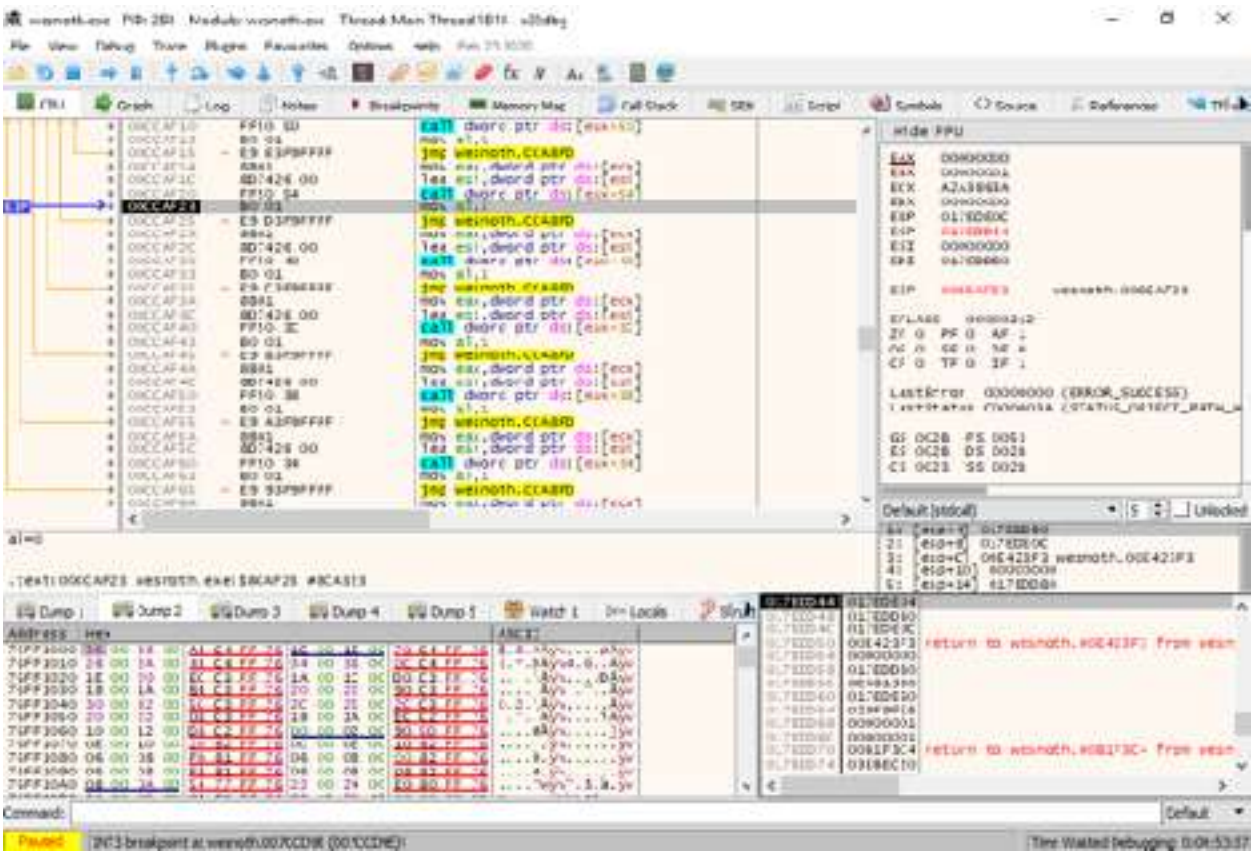
Destaque o comando de chamada no topo da linha é a chamada que acabamos de entrar. Nosso código atual é responsável por chamar esta função. Podemos usar essa tecnologia para produzir continuamente recursos com os quais nos importamos.

Sabemos que funções que processam menus com botão direito têm muitos ramos e chamadas. Podemos adivinhar que, quando traduzida para uma compilação, a frase de troca do jogo provavelmente será assim:

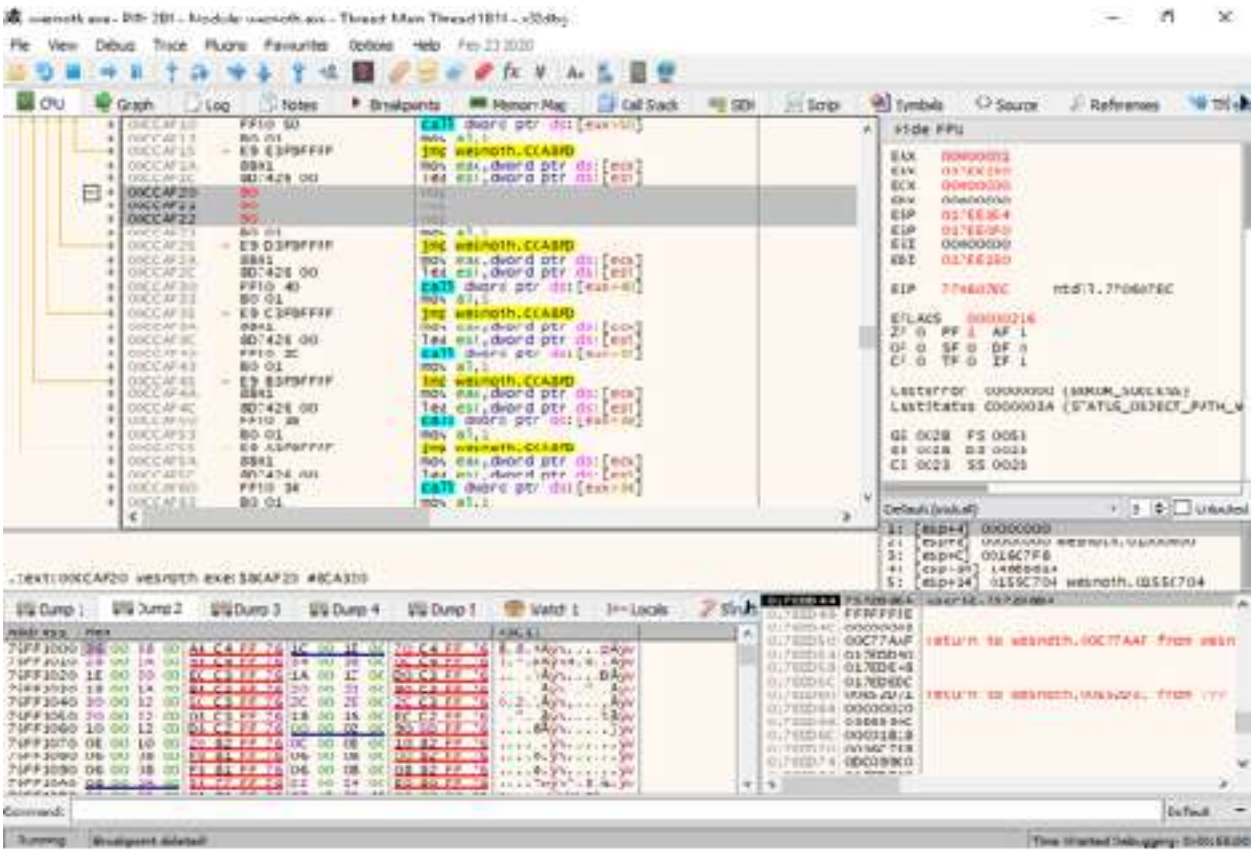
```
Chamar some_address
jmp para _end
Chamar some_address
jmp para _end
Ligue para alguns endereços
```

```
jmp para _end
```

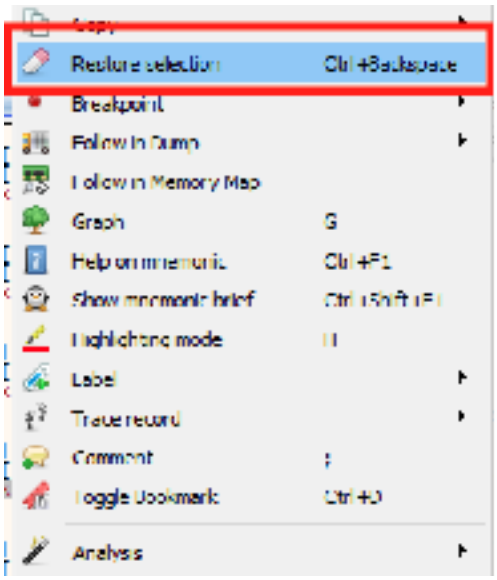
É provável que haja outras explicações, mas este é o formato que estamos procurando . Continue seguindo o loop de execução até que a instrução seja retornada e depois saia. Depois de várias vezes, você deve pousar no seguinte código:



Esse padrão parece semelhante ao que esperávamos. Podemos verificar que este é o código correto cancelando a chamada da qual acabamos de sair, conforme mostrado abaixo:



Se você voltar ao jogo e tentar recrutar uma unidade, nada acontecerá. Isso é uma boa prova de que encontramos uma função responsável por lidar com eventos de menu com botão direito no RCruiting. Volte a x64dbg, clique com o botão direito no código que acabamos de alterar e selecione Restore selection. Isso restaurará a instrução original.

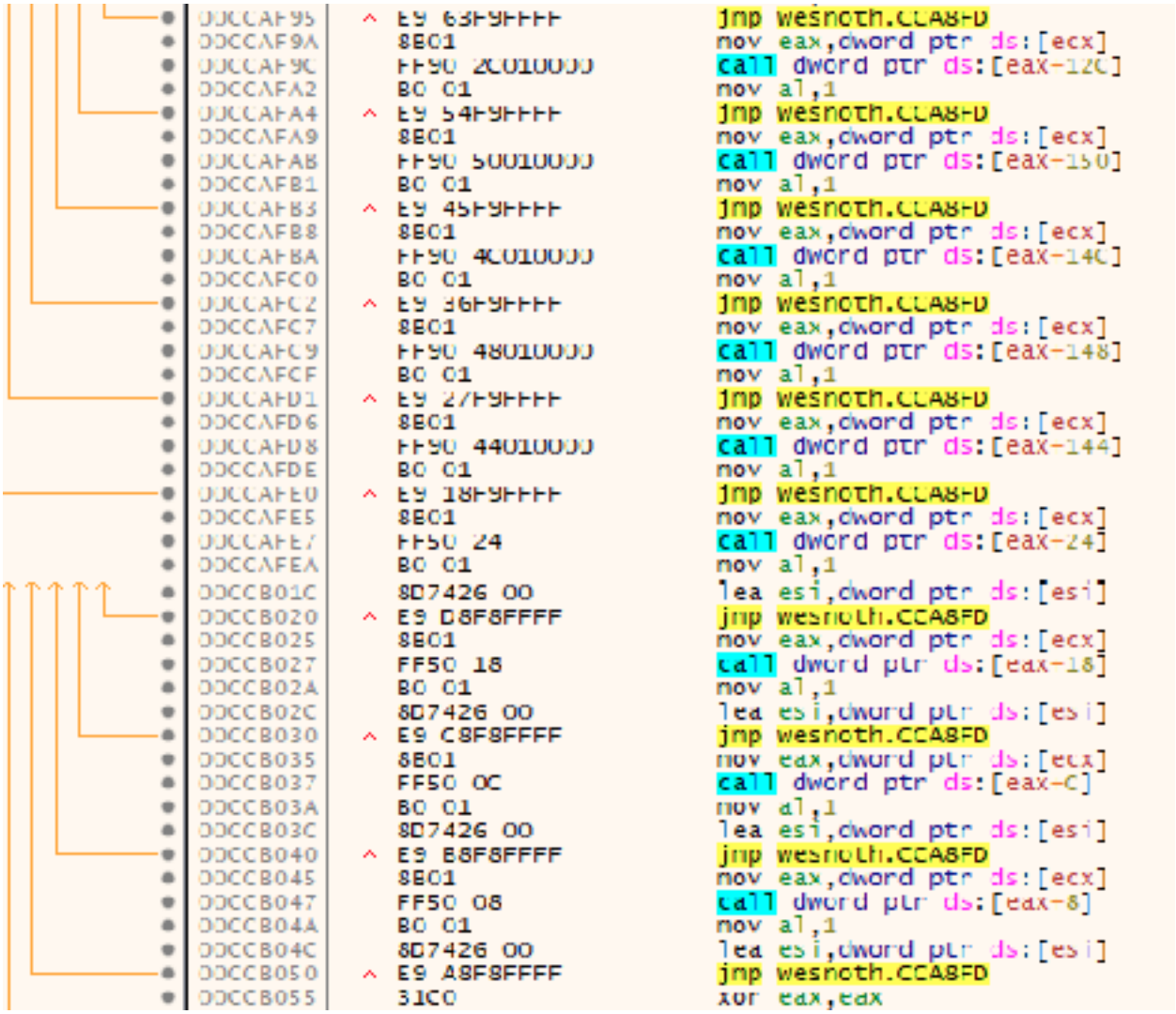


2.4.7 Localização de outros eventos

Agora que encontramos chamadas para eventos de recruit, podemos usar sua estrutura para calcular como outros eventos no jogo são chamados. O telefone parece:

```
Chamar dword ptr ds: [eax+0x54]
```

Esta chamada não é uma chamada para uma posição estática. Em vez disso, ele chama a posição eax+0x54 na memória. Se olharmos para outras chamadas na função, veremos que todas elas têm formas semelhantes, com apenas o último número mudando.

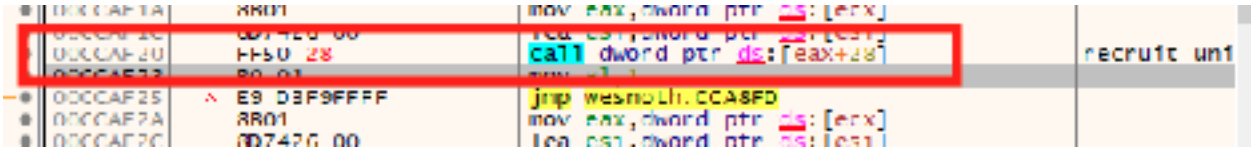


Devido a essa estrutura, temos que modificar o modelo de código original com a declaração de mudança. Na captura de tela acima, podemos ver que o último número é sempre um múltiplo de 4. Portanto, podemos presumir que essas funções provavelmente serão armazenadas em algum tipo de lista ou matriz. O código original do jogo pode ser semelhante a:

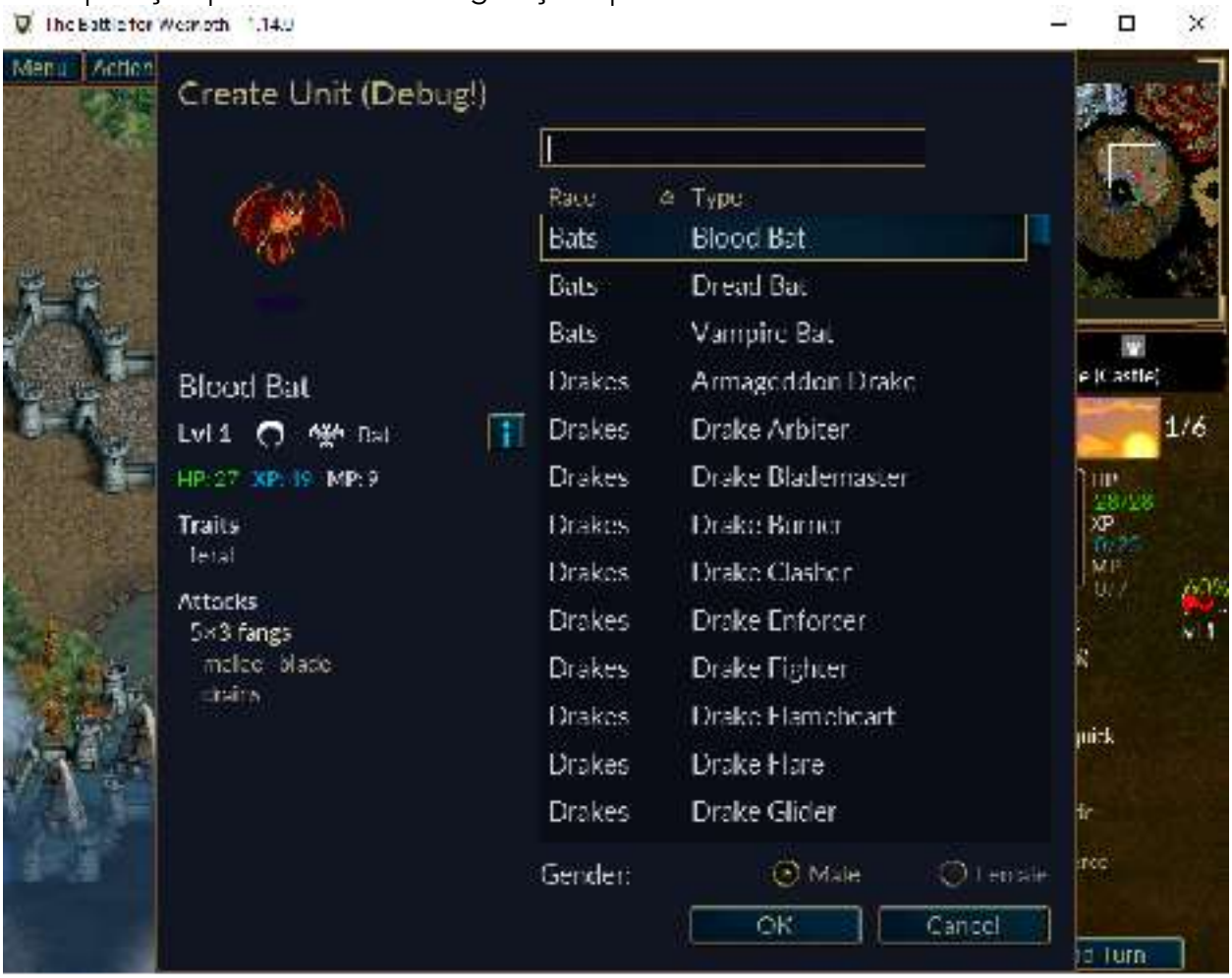
```
void*context_menu_funções[max_funções]={terrain
_descripcionações,
Unidade de recrutamento,
...
}

Context_menu_funções[opcion_selecionado]();
```

Este código armazena um ponteiro apontando para cada função no array. Você pode então usar a variável option_selected para recuperar a função correta do array e executá-la. Discutiremos ponteiros em capítulos posteriores. É importante observar que, mesmo que tenhamos códigos brutos errados em nossas mentes, a estrutura geral dos ramos é sempre óbvia no código do jogo.

Sabemos que o deslocamento de recrutamento é 0x54. Para determinar o deslocamento adicional, podemos alterar a chamada de recrutamento para um valor diferente e registrar os resultados ao usar a entrada de recrutamento no menu de contexto. Começando com eax, podemos tentar cada múltiplo de 4 e registrar seus resultados (eax+4, eax+8, eax+0xc, eax+0x10, etc.). Por exemplo, ao alterar o valor para 0x28, uma descrição do terreno será exibida quando tentamos recrutar uma unidade.



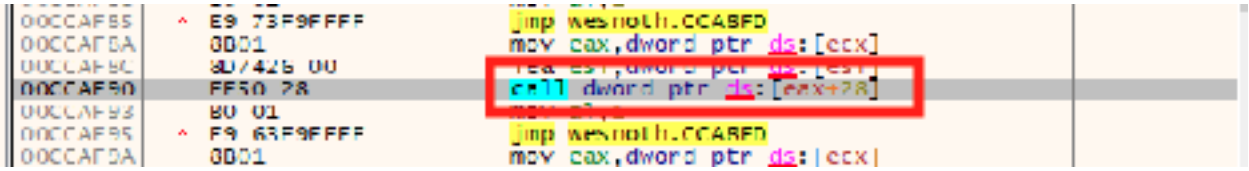
Ainda mais interessante é quando mudamos o valor para 0x68. Neste caso, um menu de depuração para a unidade de geração aparecerá.



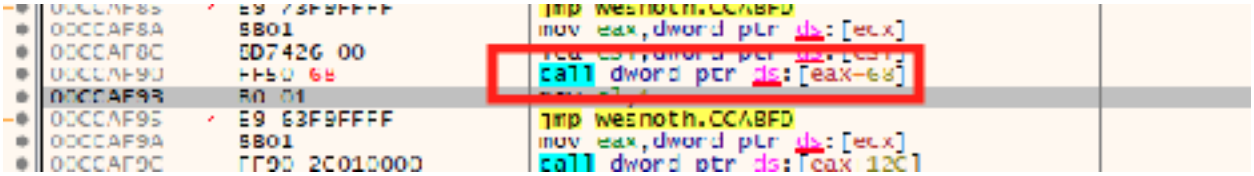
2.4.8 Alteração

Podemos criar nossos hackers usando os dois valores encontrados acima. Primeiro, encontraremos o código do item de menu responsável por exibir a descrição do terreno. Em seguida, alteraremos o valor para chamar o menu de depuração.

Sabemos que o valor da descrição do terreno é 0x28. Ao observar a área ao redor do call recruit, eventualmente encontramos o código responsável por descrever o evento do terreno.



Em seguida, mudaremos esse valor para 0x68. Isso chama o menu de depuração sempre que selecionamos uma descrição do terreno. Como as descrições do terreno podem ser encontradas em qualquer azulejo, isso nos permitirá recrutar unidades em qualquer lugar.



Após concluir esta alteração, retorne a Wesnoth, selecione um bloco aleatório e selecione uma descrição do terreno. Selecione uma unidade do menu de depuração e verifique se o hacker é válido.

2.5 Código Caverna

2.5.1 Histórico

Nos dois primeiros capítulos, modificamos o código do jogo para alterar sua funcionalidade. Para essas duas mudanças, substituímos a instrução original por uma nova instrução. Mas e se quisermos manter a instrução original ou substituí-la por várias instruções? Nesses casos, precisamos usar a caverna de código.

O código caverna faz parte da memória do jogo, e nós o preenchemos com comandos . Então mudamos o código original do jogo para chamar essas instruções. O nome vem do fato de que estamos criando uma "caverna" de comandos ocultos. A maioria dos jogos tem uma grande quantidade de memória não utilizada entre funções ou no final de arquivos executáveis. Esses locais são perfeitos para criar cavernas de código.

2.5.2 Redirecionamento

No capítulo anterior, mudamos a função que exibe a descrição do terreno para chamar o menu de depuração. Usando o Code Cave, ainda podemos chamar o menu de depuração, mas também podemos chamar a função de descrição do terreno posteriormente. Ao fazer isso, não perderemos nenhuma função do jogo.

As instruções originais para a chamada de descrição do terreno são as seguintes:

```
0x00CCAF90 chamada dword ptr ds: [eax+28]
```

Para este exemplo, suponha que haja um segmento de memória vazio em 0x00D00000. Nosso primeiro objetivo foi recriar a chamada original em 0x00D00000 e redirecionar o código original para esse novo código. Primeiro, copiamos a instrução original para

0x00D00000:

```
0x00D00000 Chamada dword ptr ds: [eax+28]
```

Em seguida, redirecionaremos o código original para esta chamada:

```
0x00CCAF90 jmp 0x00D00000
```

Por fim, em nossa caverna de código, precisamos retornar ao código original. Isso pode ser alcançado saltando para a instrução que substituímos. Neste caso, o próximo comando no jogo é 0x00CCAF93. A caverna de código que completamos parece:

```
0x00D00000 Chamada dword ptr ds: [eax+
28]jmp 0x00CCAF93
```

2.5.3 Descrição de Restauração

Até agora, esta caverna de código apenas reconstruiu as instruções originais. Este é um primeiro passo importante para garantir que nossa redireção não interrompa nada. Isso nem sempre acontece, especialmente ao lidar com funções de jogo que modificam pilhas. Discutiremos como lidar com essas questões em capítulos posteriores. Agora, observe que redirecionar o código do jogo nem sempre é um processo tranquilo.

Ao escrever uma caverna de código, é crucial modificar apenas o que você precisa, sem alterar o resto. Alterações inesperadas em outros registros, partes de memória ou pilhas podem causar a falha do jogo. Para ilustrar esse princípio, suponha que tenhamos o seguinte código para o qual pretendemos redirecionar:

```
mov eax, 999
chamada 0xDEADBEEF
```

Suponha que redirecionemos a chamada para uma caverna de código, conforme mostrado abaixo:

```
mov eax, 123
Chamar some_other_função
chamada 0xDEADBEEF
jmp back
```

Se a função 0xDEADBEEF exigir um eax de 999, o jogo lançará uma exceção e entrará em colapso. Embora este seja um exemplo insignificante, chamar uma função de jogo geralmente tem muitos efeitos colaterais que você não percebe.

Para salvar e restaurar os valores de registro do jogo (eax, ebx, ecx, etc.), usaremos dois comandos: Pusha e popad. Pushaad empurra (ou salva) todos os valores de registro na pilha. O popad exibe (ou recupera) todos os valores de registro da pilha. Em capítulos subsequentes, apresentaremos a pilha em si e como restaurar a pilha do jogo. No entanto, restaurar o valor do registro evitará que a maioria das falhas sejam afetadas.

2.5.4 Esqueleto da Caverna

Com essas instruções, agora temos a estrutura básica da caverna de código. Parece:

```
Pushad
Executar o novo recurso popad

Chamar a instrução original
jmp de volta ao código do jogo
```

Voltemos ao exemplo de Wesnoth. Na Seção 2.5.2 acima, temos as seguintes cavernas de código:

```
0x00D00000 Chamada dword ptr ds: [eax+
28]jmp 0x00CCAF93
```

Com Pusha/popad, agora podemos introduzir novas instruções com segurança. O código para chamar o menu de depuração é o seguinte:

```
Chamar dword ptr ds: [eax+68]
```

Vamos assumir que a chamada não modifica a pilha de forma alguma. Podemos chamar essa função com segurança na caverna de código salvando o registro, chamando a função e , em seguida, restaurando-a. Em seguida, executaremos a instrução original e retornaremos ao código do jogo após a instrução original. Nossa caverna final de código parece:

```
0x00D00000 push
Chamar dword ptr ds: [EAX 68]popad

Chamar dword ptr ds: [eax+28]jmp 0x00
CCAF93
```

Ao fazer isso, criamos um buraco no código do jogo, substituindo um comando por vários comandos. Desde que todo o conteúdo seja restaurado, não há limite para a quantidade de novo código que pode ser chamado.

2.6 Código de uso cavernas

2.6.1 Objetivo

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

2.6.2 Identificação

Nosso objetivo neste capítulo é criar uma caverna de código dentro do Wesnoth. Sempre que selecionamos uma descrição do terreno, esta caverna de código é executada. Antes de chamar a caixa de descrição do terreno, a Code Cave nos dará 999 moedas de ouro.

2.6.3 Compreensão

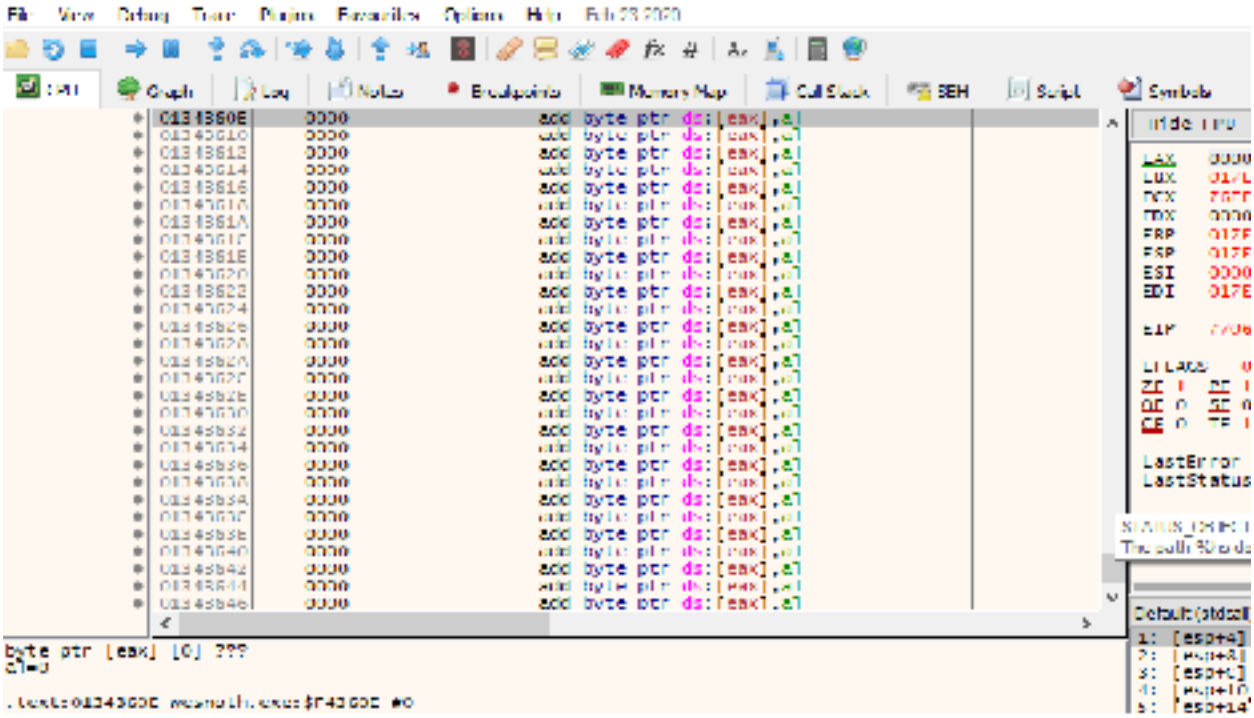
Para criar uma caverna de código, primeiro precisamos encontrar dois locais: o local para redirecionar e o local para colocar o esqueleto da caverna. Em seguida, criaremos um esqueleto de caverna no segundo local. Uma vez criado, redirecionaremos a primeira posição para pular para o esqueleto.

2.6.4 Posicionamento de Ouro

Como usaremos esse hacker para modificar nosso ouro, precisamos encontrar nosso endereço de ouro . Nosso primeiro passo foi abrir o Wesnoth e criar um jogo local. Em seguida, seguiremos os passos do Capítulo 1.5 para encontrar nosso endereço dourado. Conforme discutido no Capítulo 2.3, nosso endereço dourado estará localizado em um endereço diferente dos capítulos anteriores. Assim que encontrarmos um novo endereço dourado, desligaremos o mecanismo de trapaça, mas manteremos o Wesnoth aberto. Neste capítulo, usaremos o valor 0x05F3B85C como nosso endereço áureo.

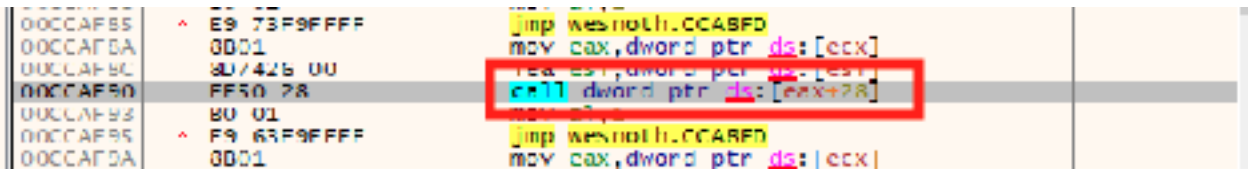
2.6.5 Buraco de posicionamento

Primeiro, encontraremos onde colocar a caverna de código. Embora possamos encontrar espaços em branco na memória em muitos lugares, a maneira mais rápida e fácil de fazer isso é rolar até o final do módulo Wesnoth. No final da maioria dos módulos executáveis, há um grande pedaço de dados vazios que podem ser modificados. No nosso exemplo, essa memória é aproximadamente 0x0134360E.



2.6.6 Posição do gancho

Em seguida, precisamos identificar o endereço a ser vinculado. Neste capítulo, faremos ganchos para mostrar como o terreno é descrito. Para encontrar esse endereço, podemos usar o mesmo método e endereço que determinamos no Capítulo 2.4. Esta chamada está localizada em 0xCCAF90.



2.6.7 Redirecionamento

Depois de determinar a localização da caverna de código e do gancho, agora podemos criar a caverna de código. É importante que o jogo esteja suspenso ao modificar o código do jogo no depurador. Isso evita que a execução normal do jogo entre acidentalmente em nossa caverna de código antes que possamos terminá-lo. Se for esse o caso, o jogo pode pular para um código inexistente e travar. Para pausar o jogo, podemos usar o botão de pausa ao lado do botão Continuar.

Agora podemos redirecionar a posição do gancho para pular para nossa caverna de código. No Capítulo 1.1, apresentamos o conceito de código de operação. Cada instrução tem um código de operação diferente e diferentes comprimentos de código de operação. Isso ocorre porque certas instruções exigem dados adicionais para serem executadas. Por exemplo, as instruções pushad são representadas por um código de operação de 1 byte 0x60. Isso ocorre porque as instruções não exigem dados adicionais para serem executadas. Em vez disso, o comando mov al, 1 é representado pelo código de operação de 2 bytes 0xB0 01. Isso ocorre porque as instruções mov exigem que os dados movidos (neste caso, o valor 1) sejam codificados em algum lugar dentro do código de operação.

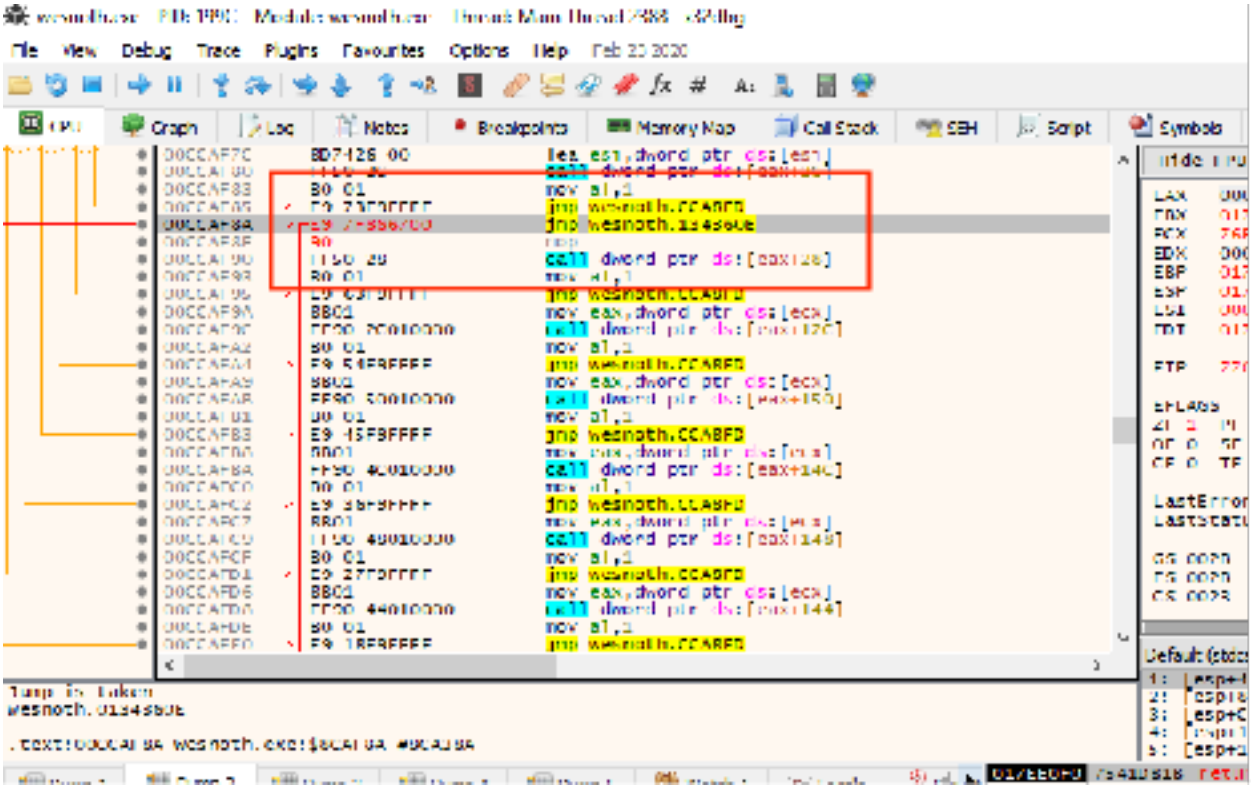
O código de operação da instrução jmp é de 5 bytes de comprimento. Isso ocorre porque o código de operação codifica o endereço para o qual será redirecionado. Portanto, precisamos encontrar um local de pelo menos 5 bytes para colocar nossas instruções jmp. Para isso, vamos examinar como exibir descrições do terreno:

```
Instruções do código de operação--

8B01 mov eax, dword ptr ds: [ecx]8D7426 00
LEESI, dword ptr ds: [esi]FF50 28 Chamar
dword ptr ds: [eax+28]B0 01 mov al, 1
```

Infelizmente, esse método não contém instruções com um código de operação de 5 bytes. Neste caso, precisamos substituir as duas primeiras instruções. Ao escrever uma caverna de código, precisamos lembrar de substituir esses dois. Entretanto, os códigos de operação das duas primeiras instruções (0x8B01 e 0x8D742600) combinados têm um total de 6 bytes. Quando substituímos os primeiros 5 bytes por um salto, o último byte (0x00) será mantido e pode ser

Execução. Para garantir que nossas mudanças não façam com que o jogo caia, substituiremos o último byte por instruções nop. O x64dbg faz isso automaticamente ao montar instruções.

Com tudo isso, finalmente podemos mudar nosso código. Navegue para o primeiro comando mov em 0x00CCAF8A e altere esse comando para jmp 0x0134360E. Ao montar este manual, certifique-se de selecionar a opção "Preencher com NOPs". Este salto será responsável por pular em nossa caverna de código.

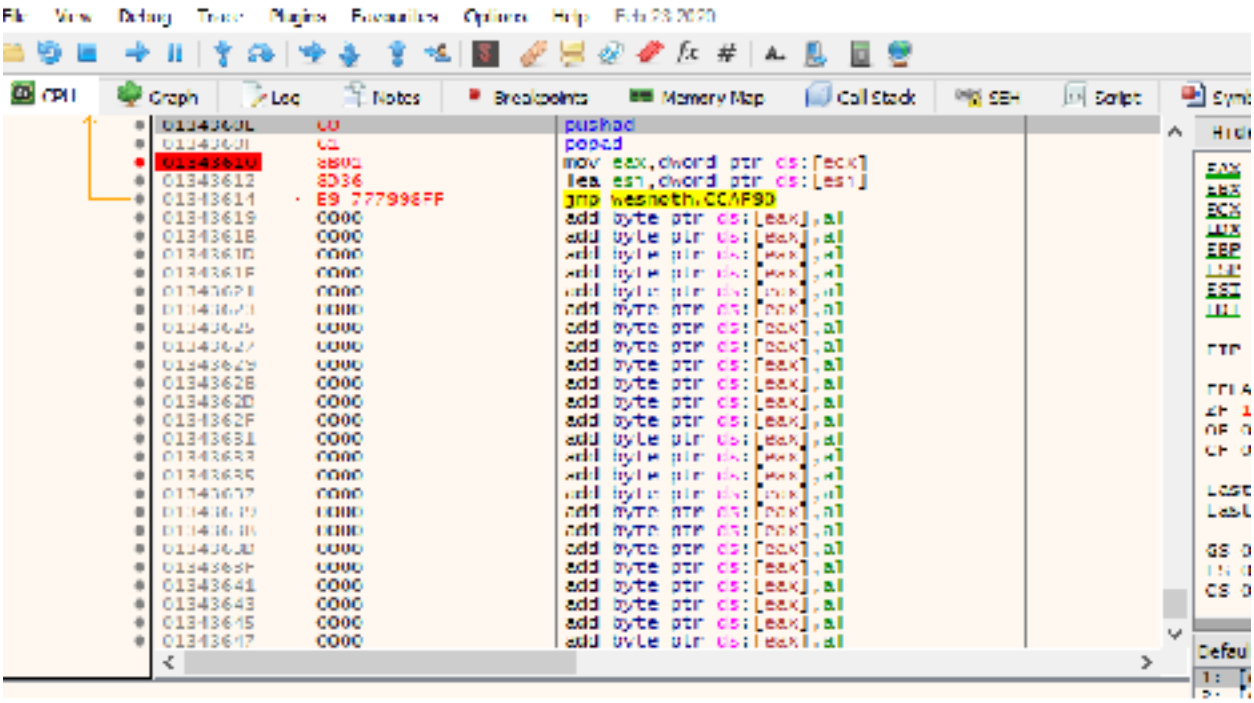


2.6.8 Esqueleto da Caverna

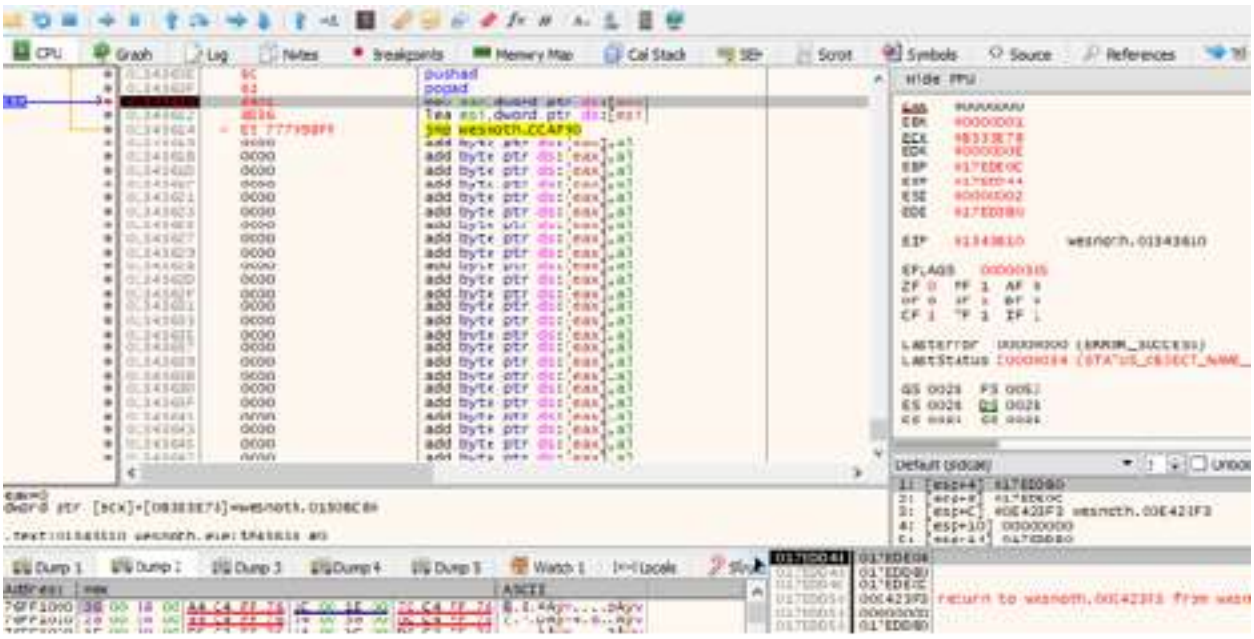
Em seguida, podemos escrever sobre o esqueleto da nossa caverna. Agora, nosso esqueleto da caverna simplesmente salvará e restaurará os registros, substituirá as instruções originais e, em seguida, retornará ao código original. Isso será executado da mesma maneira que o código original e nos permitirá verificar se nossa redireção foi bem-sucedida. Navegue para 0x0134360E e insira o seguinte código:

```
Pushad
Rebote
mov eax, dword ptr ds: [ecx]
```

```
LeESI, dword ptr ds: [esi]jmp
0xCCAF90
```



Para verificar que este código cave está sendo chamado, coloque um ponto de interrupção em 0x01343610 e continue executando o programa. Quando você chamar a descrição do terreno, os pontos de interrupção em nossa caverna de código devem aparecer.

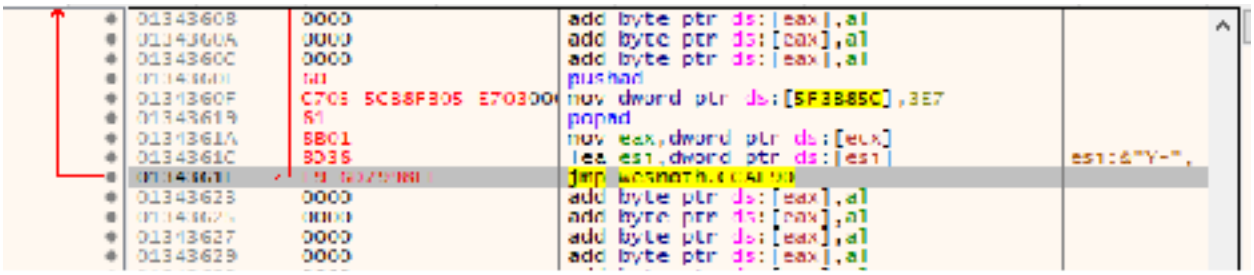


2.6.9 Alteração

Com o esqueleto de trabalho, agora podemos mudar nosso ouro. Entre as instruções Pusha e popad, inserimos uma instrução para modificar o gold. Para isso, movemos o valor de 999 para o endereço onde o valor dourado é armazenado. Esta instrução é a seguinte:

```
mov dword ptr ds:[0x5F3B85C],0x3E7
```

Com esta instrução, nossa caverna de código final será a seguinte:



Se voltarmos ao Wesnoth e selecionarmos a descrição do terreno no azulejo, nosso ouro ficará 999 antes que a moldura da descrição do terreno apareça.

2.7 Memória Dinâmica Alocação

2.7.1 Visão geral

Nos capítulos anteriores, modificamos o ouro do jogador em Vistanos. Sempre que reiniciamos o jogo, temos que repetir o processo de encontrar o endereço de memória dourada do jogador, porque cada vez é diferente. Isso se deve à alocação dinâmica de memória, ou DMA. Para escrever hackers que podem ser reutilizados e distribuídos, precisamos converter esses endereços "aleatórios" em endereços consistentes de alguma forma. Há muitas maneiras de fazer isso, mas primeiro precisamos discutir como o DMA funciona e por que ele existe.

2.7.2 Histórico

Como discutimos no Capítulo 1.2, jogos são programas grandes com recursos significativos. Não é possível colocar todos os dados do jogo na RAM de uma só vez, então eles devem ser carregados quando necessário. Por exemplo, um jogo não carregará modelos ou imagens de inimigos até que o jogador esteja prestes a encontrá-los. Esse processo é chamado de carregamento dinâmico de recursos.

Esses recursos carregados dinamicamente devem ser colocados em alguma parte da memória para que o jogo possa acessá-los novamente. O jogo é responsável por criar e destruir essas partes de memória. Criação é chamada de distribuição e destruição é chamada de libertação. Portanto, a alocação dinâmica de memória é o processo de criação de seções de memória para salvar recursos quando o jogo precisa de recursos. OS jogos só podem solicitar memória do sistema operacional e não podem controlar a localização dessa memória.

Vamos considerar o ouro do jogador em Vistanos e como ele foi criado. Quando o Wesnoth é ativado, o perfil do jogador é carregado na categoria Player do jogo. Os jogadores podem então escolher entre vários modos de jogo e outras opções no menu principal. Se um jogador então iniciar um jogo, vários itens são atribuídos e colocados na categoria do jogador, como a raça do jogador, suas unidades disponíveis e seu ouro. Quando o jogador sai do jogo,

Então esses valores são destruídos. É por isso que o endereço dourado dos jogadores é sempre diferente.

2.7.3 Programação

Para escrever um hacker, precisamos de algumas maneiras de encontrar endereços dourados de forma consistente

Pesquise no mecanismo de trapaça. Há várias maneiras de fazer isso:

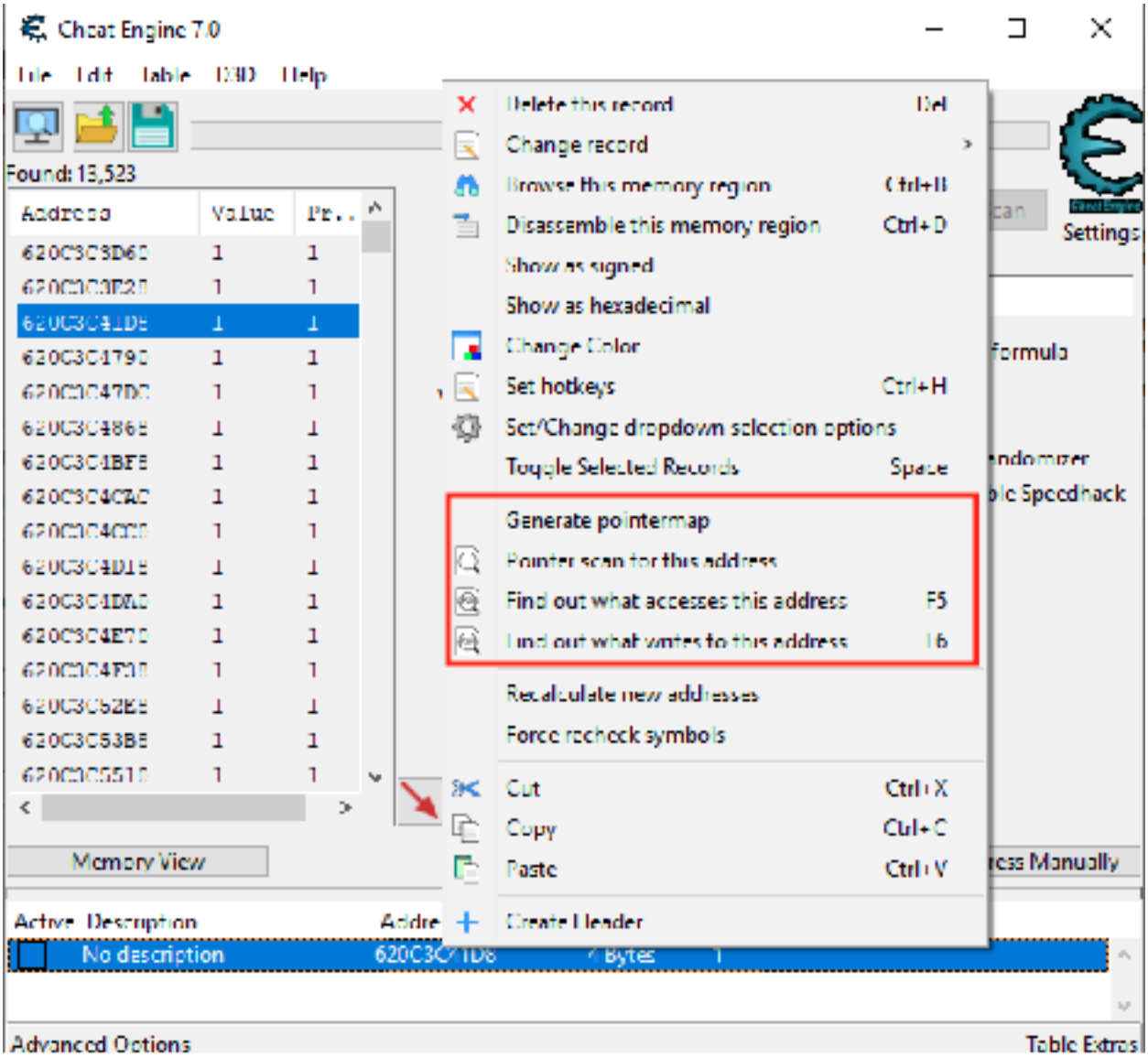
- Scanners automáticos, como mecanismos de trapaça
- Caverna de Código
- Ré

Esses métodos podem ser usados para encontrar qualquer endereço dinâmico. Discutiremos brevemente cada abordagem neste capítulo, mas usaremos todos eles em capítulos subsequentes.

Ao usar um mecanismo de trapaça ou reversão, nosso objetivo é encontrar algo chamado ponteiro base. Normalmente, o ponteiro base representa um endereço de memória que é sempre consistente e pode ser usado para deslocar o valor com o qual nos interessamos. Esse método funciona porque alguns endereços devem ser constantes antes que o jogo possa encontrá-los. Por exemplo, no Wesnoth, o jogo precisa saber onde está a classe do jogador. Se encontrarmos a classe Player, podemos usá-la como ponteiro base para deslocar nosso endereço dourado.

2.7.4 Mecanismo de trapaça

Um recurso do mecanismo de trapaça é a capacidade de escanear ponteiros. Isso pode ser feito encontrando um endereço (como o ouro do jogador) e clicando com o botão direito nele para acessar um menu de contexto. Este menu de contexto contém todas as funções de digitalização de ponteiros.



Este recurso retorna todos os locais de memória que atualmente se refere ao endereço selecionado. Você pode então salvar esses endereços no arquivo digitalizado. Você pode então recarregar o jogo e encontrar o endereço novamente. Em seguida, ao comparar novos locais de memória com arquivos digitalizados, apenas os locais consistentes podem ser reduzidos, semelhantes às pesquisas de memória convencionais. No final, teremos apenas um ponteiro que sempre aponta para o endereço que escolhemos.

2.7.5 Código Caverna

Outra maneira de derrotar o DMA é usar o código caverna. Dessa forma, encontre o local onde o valor esperado é acessado. Para nosso exemplo de Wesnoth, isso pode ser

Onde quer que o ouro seja trocado. Após essa posição, o código é imediatamente redirecionado para nossa caverna de código. Em nossa caverna de código, podemos salvar o valor atual em um pedaço de memória que controlamos. Essa memória pode ser acessada continuamente por nossos hackers.

Por exemplo, o código no Wesnoth que é responsável por reduzir nosso ouro ao recrutar empresas é o seguinte:

```
SubdWORD ptr ds: [edx+4], ecx
```

Quando essa instrução é executada, edx+4 contém uma referência ao endereço de memória de ouro, e ecx contém uma referência ao custo da unidade recém-recrutada. Ao redirecionar o código que segue o endereço para uma caverna, podemos salvar o valor do endereço na caverna. Aqui está uma caverna de exemplo que pode fazer isso:

```
Pushad
mov dword ptr ds: [0x12345678], edx+4
popad
...a instrução original foi substituída
... jmp 0xredikt_Localização
```

Uma vez concluído, nossos hackers podem citar 0x12345678 para obter o valor atual do endereço dourado.

2.7.6 marcha ré

A maneira final de lidar com DMA é reverter o alvo. Este método usa uma combinação dos dois métodos anteriores e é o mais comum. Nesse método, primeiro encontramos uma instrução para modificar o valor com o qual nos importamos, como a subinstrução na seção anterior. Em seguida, analisamos a função anterior à instrução e determinamos onde os registros com os quais estamos interessados (neste caso, edx) são alocados. Normalmente , isso será atribuído a um valor para outro registro com um deslocamento, como eax+60.

Em seguida, repetimos esse processo para descobrir onde o registro anterior foi alocado. Por fim, encontraremos o valor base ou ponteiro para atribuir todos esses valores. Esse ponteiro base pode então ser combinado com todos os deslocamentos que revertemos para recuperar o endereço com o qual nos interessamos.

2.8 derrotou o DMA

2.8.1 Objetivo

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

2.8.2 Identificação

O objetivo deste capítulo é encontrar o ponteiro base da classe Player. Depois, precisamos descobrir como desviar nosso endereço dourado a partir deste ponteiro de base.

2.8.3 Compreensão

Quando o jogador começa o jogo, o Wesnoth usa o DMA para atribuir vários valores, incluindo as moedas de ouro do jogador. Isso significa que o endereço dourado do jogador estará em um endereço diferente para cada jogo. Em contraste, vários valores permanecem inalterados entre os jogos, como o nome do perfil do jogador. Esses valores constantes devem ser armazenados em alguma classe de Player. Como esses valores existem em todos os jogos, deve haver um Wesnoth para localizar seus endereços estáticos. Se conseguirmos encontrar esse endereço estático, podemos deslocá-lo para nosso endereço dourado dinâmico no jogo.

Para visualizar isso, vamos imaginar a classe Players da Wesnoth como é a seguinte:

```
Jogador {
Player_Nome="IEUser"; int wins=100

Jogo = vazio;}
```

Os jogos parecem:

```
Jogos
semelhantes a {
String Edge; int
Good; int turn
}
```

Quando um jogador entra no jogo, o código do jogo aloca memória para o objeto do jogo e todos os valores que ele contém:

```
Jogadores. Jogo = Jogo novo ("Humano", 100, 1);
```

Ao encontrar o valor do endereço dourado, podemos reverter o jogo para encontrar o valor do endereço do jogo atual. Podemos então usar esse endereço para encontrar o endereço da classe Player. Como a classe Player é sempre carregada, ela será um endereço consistente. Na classe Player, podemos usar o endereço encontrado durante a reversão para deslocar o endereço dourado atual.

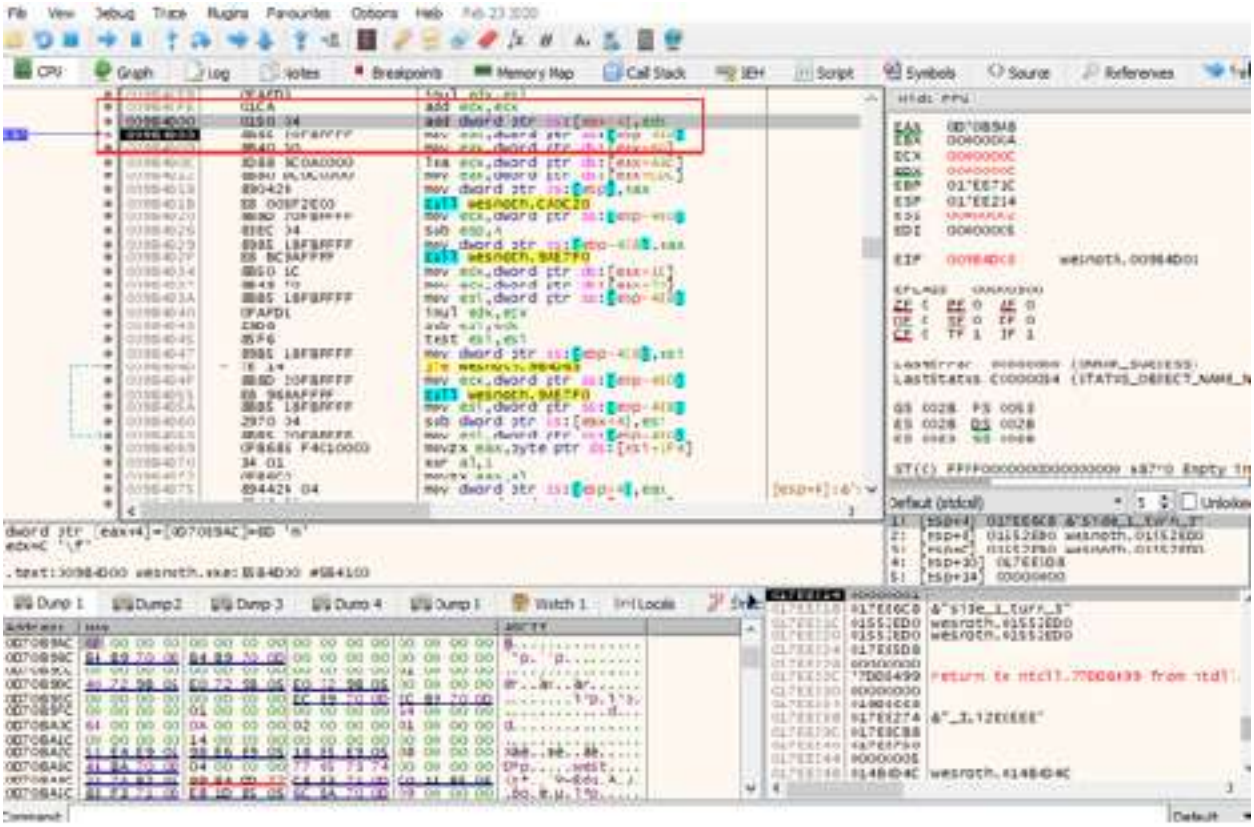
2.8.4 Posicionamento de Ouro

Na última vez deste livro, precisamos encontrar o endereço do nosso valor dourado. Nosso primeiro passo foi abrir o Wesnoth e criar um jogo local. Ao contrário dos capítulos anteriores, certifique-se de ganhar dinheiro para facilitar o processo de reversão. Além disso , certifique-se de que o segundo jogador seja definido como um oponente de computador.

Em seguida, siga os passos do Capítulo 1.5 para encontrar seu endereço dourado. Assim que você encontrar seu novo endereço dourado, desligue o mecanismo de fraude, mas mantenha Westminster aberto.

2.8.5 ponteiros de base

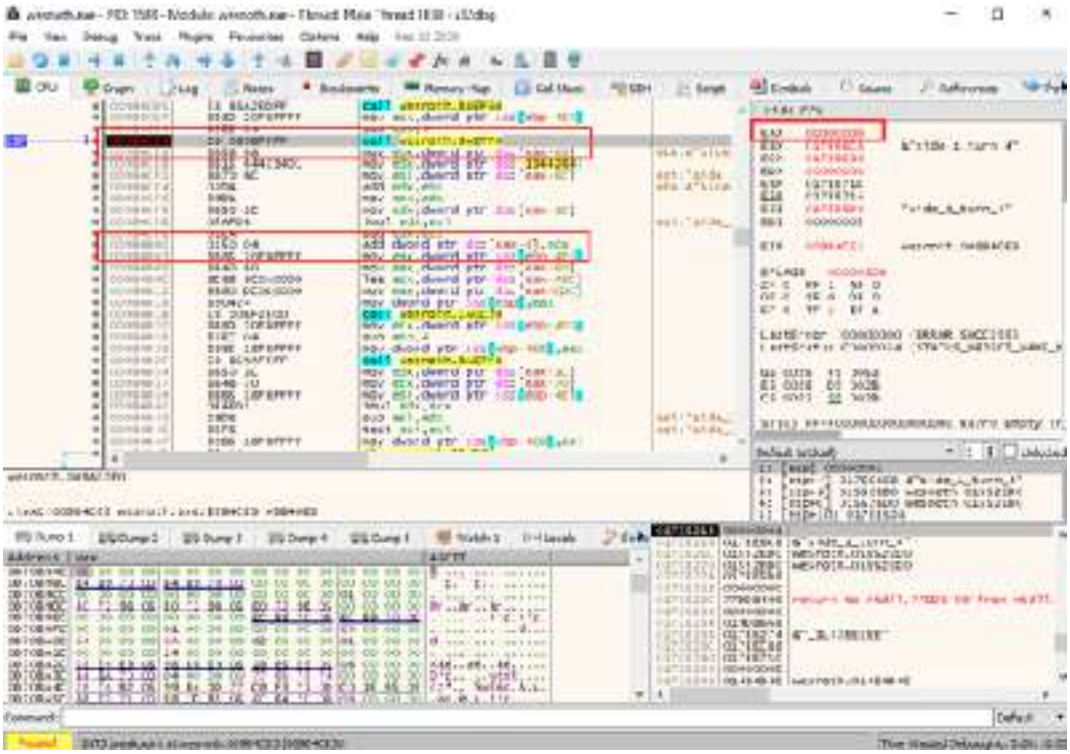
Em seguida, adicione x64dbg ao Wesnoth e defina o ponto de interrupção de gravação no endereço dourado que você encontra. Ao contrário dos capítulos anteriores, não recrute uma única unidade. Em vez disso, escolha encerrar seu turno. No final da sua rodada, seu ponto de interrupção deve sair à medida que a renda é adicionada à sua moeda de ouro.



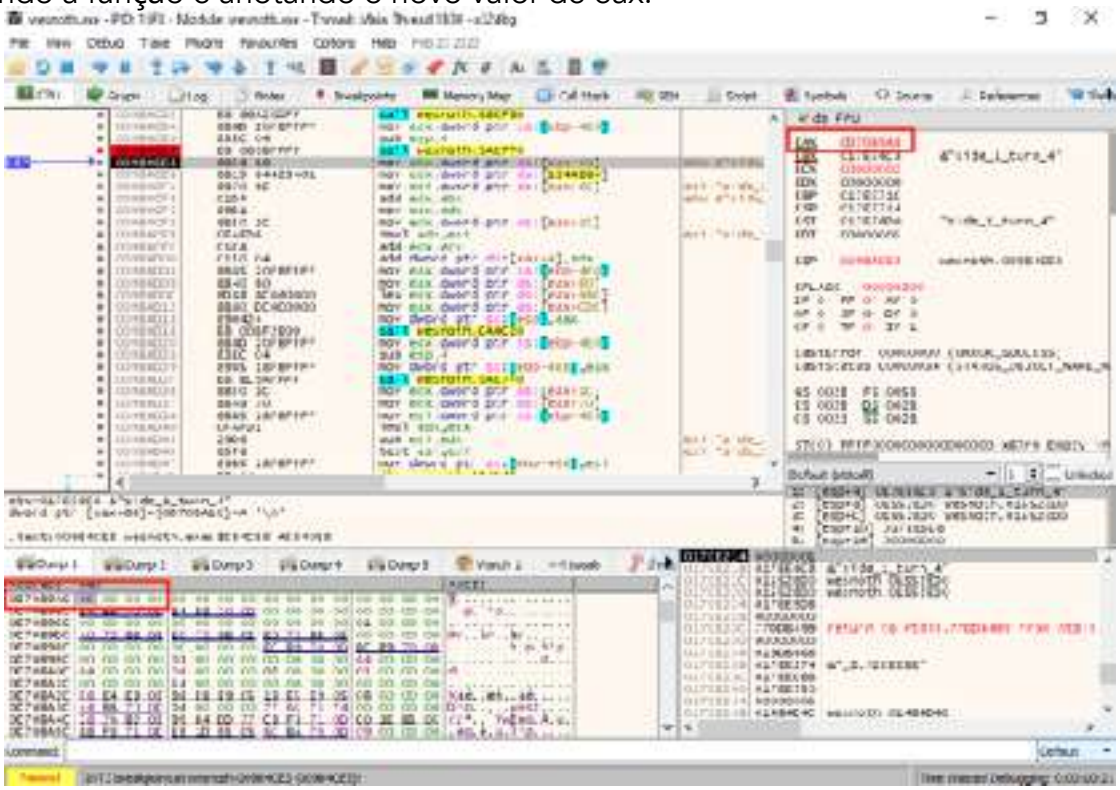
Vamos dar uma olhada rápida nas instruções que aparecem no ponto de interrupção:

```
009B4D00 Adicionar dword ptr ds: [eax+4], edx
```

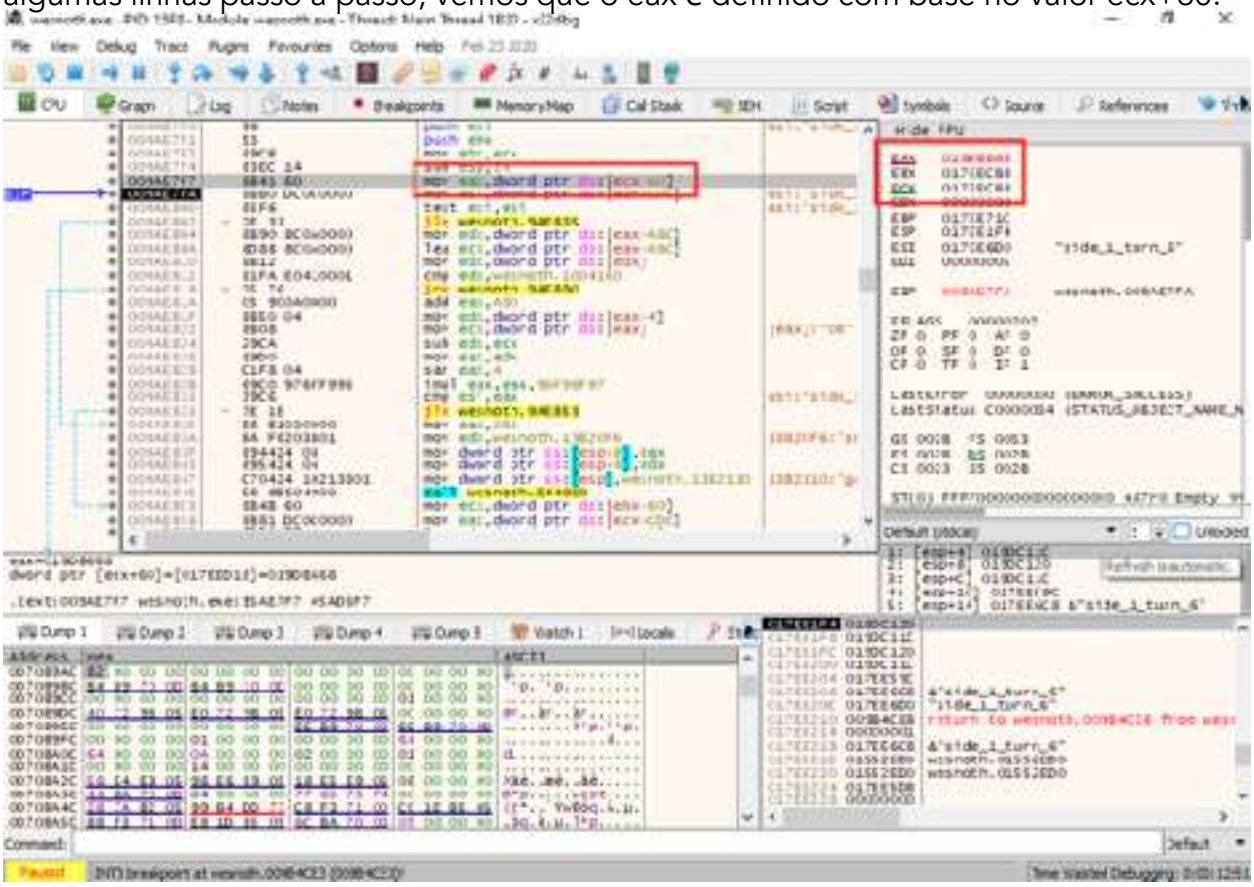
Ao executar esta instrução, eax+4 salva o valor do nosso endereço dourado (neste caso, 0x0D70B 9AC). Nosso próximo passo é determinar como alocar eax. Se olharmos para o topo do comando AD, veremos vários comandos mov que se referem ao valor eax. Acima disso, temos uma instrução de chamada para uma função desconhecida. Para determinar se essa função é responsável por definir eax, podemos definir um ponto de interrupção nesta chamada e continuar Wesnoth. Esse ponto de interrupção será atingido quando terminarmos a rodada novamente.



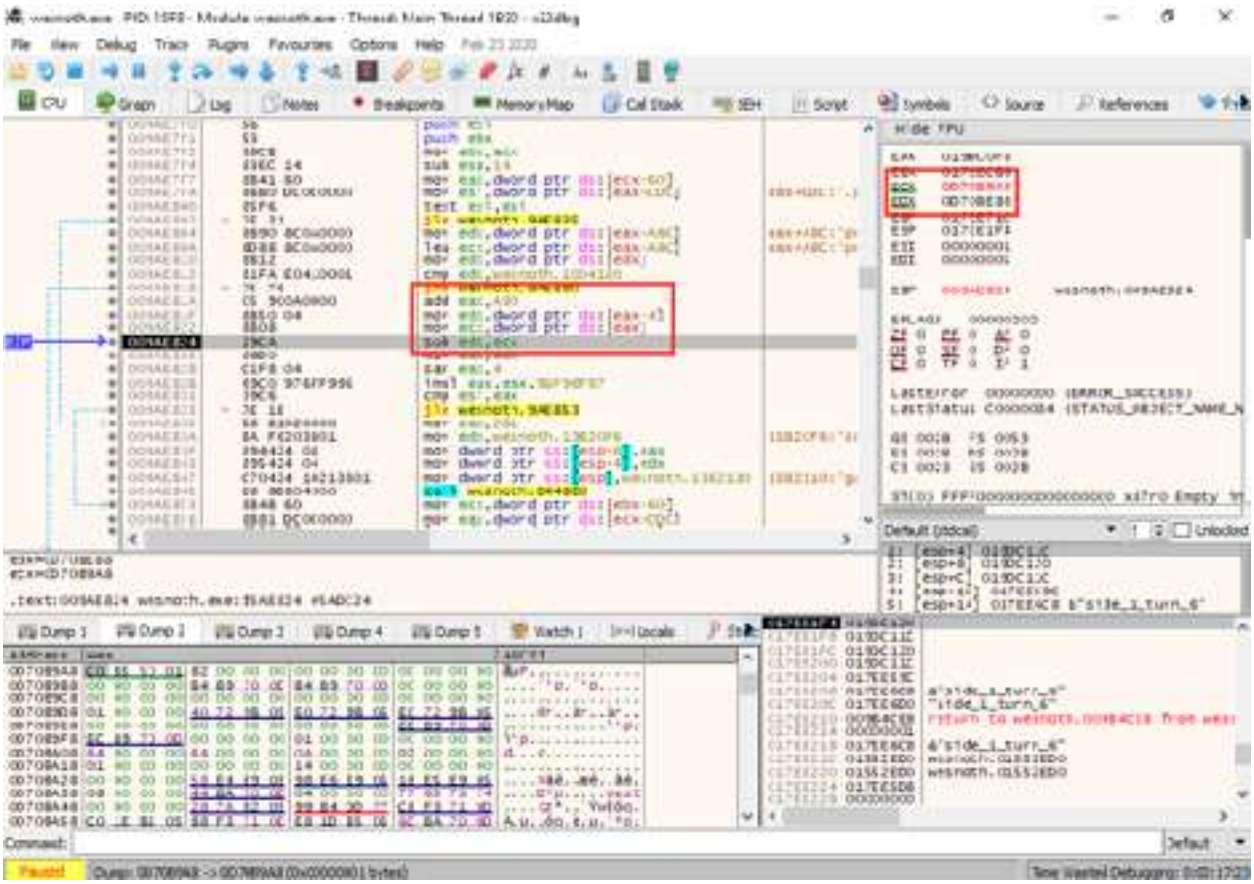
Observando o registro, vemos que eax é definido para 0 ao entrar na função, o que significa que a função deve obter o valor correto de eax. Podemos confirmar isso pulando a função e anotando o novo valor de eax.



Após a confirmação, continue Wesnoth e encerre sua rodada novamente para acionar o mesmo ponto de interrupção. Desta vez, entre na função. Depois de completar algumas linhas passo a passo, vemos que o eax é definido com base no valor ecx+60.



Agora, notaremos que precisamos determinar o valor do ecx. Entretanto, o eax ainda não está próximo do valor do nosso endereço dourado. Antes de continuar determinando o valor do ecx, precisamos determinar como o eax é modificado da atribuição inicial para atingir o endereço áureo. Se continuarmos a funcionar para baixo, veremos o valor de 0 xA90 adicionado ao eax. Depois disso, o valor de carga edx é eax+4. Vamos para o endereço atrás do código e vejamos quais valores são carregados em edx.



Podemos ver aqui que o valor do edx está a apenas 4 de distância do nosso endereço áureo , da mesma maneira que a instrução original do add o referiu. A partir disso, sabemos que [[ecx+60]+0xA90]+4 é nosso endereço áureo. Nosso próximo passo é determinar o ecx.

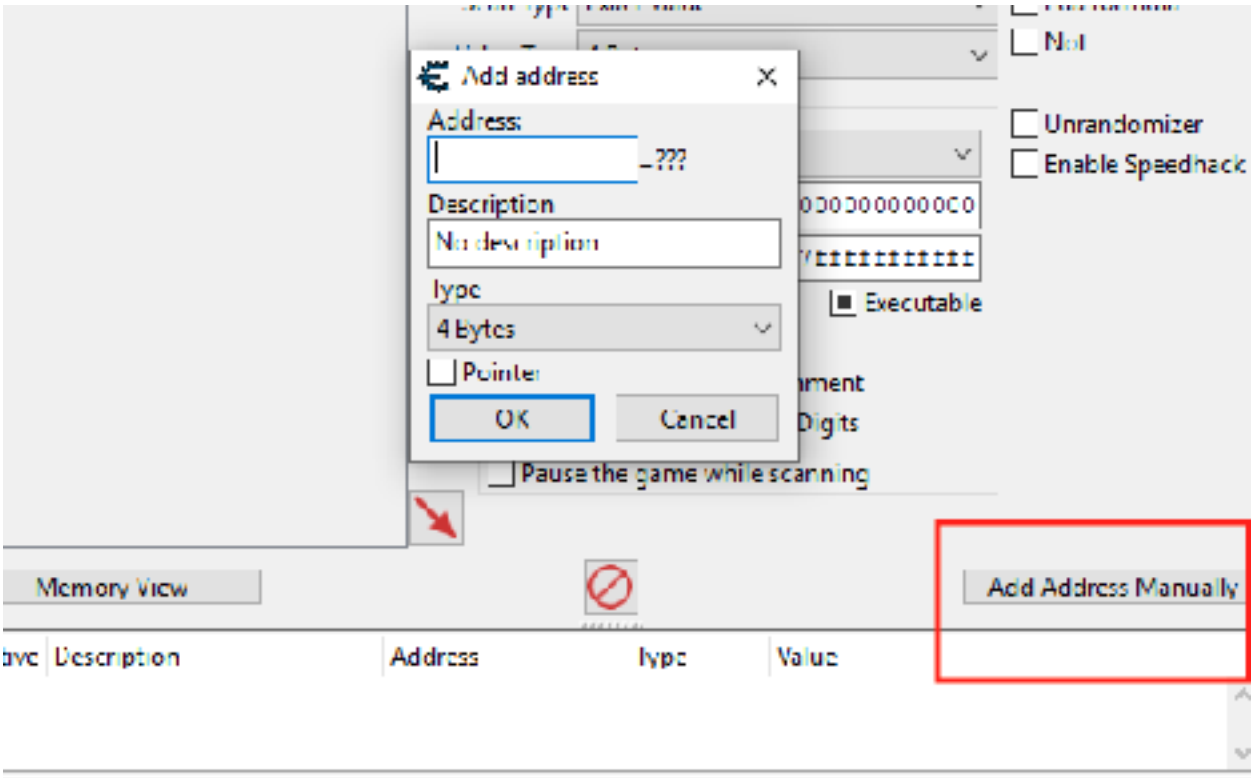
Ao localizar o ponteiro de base, é importante parar toda vez que um novo registro ou endereço for introduzido e garantir que ele seja realmente aleatório. Para isso, primeiro registre o valor atual quando o ecx é carregado no eax. Neste exemplo, o valor é 0x017 EECB8. Em seguida, anote o endereço da instrução atribuída para que você possa colocar outro ponto de interrupção aqui. Neste caso, estará em 0x009AE7F7.

Agora, separe o x64dbg e desligue o Wesnoth. Uma vez desativado, reinicie o Wesnoth novamente , recrie um jogo com as mesmas configurações de receita e conecte-se novamente ao x64dbg. Coloque um ponto de interrupção no endereço indicado (0x009AE7F7) e encerre sua rodada. Quando o ponto de interrupção aparecer, observe o novo valor do ecx. Neste caso, é exatamente o mesmo da última vez (0x017EECB8). Portanto, sabemos que isso deve representar um ponteiro de base invariável. Se isso realmente mudar, teremos que continuar o processo de reversão.

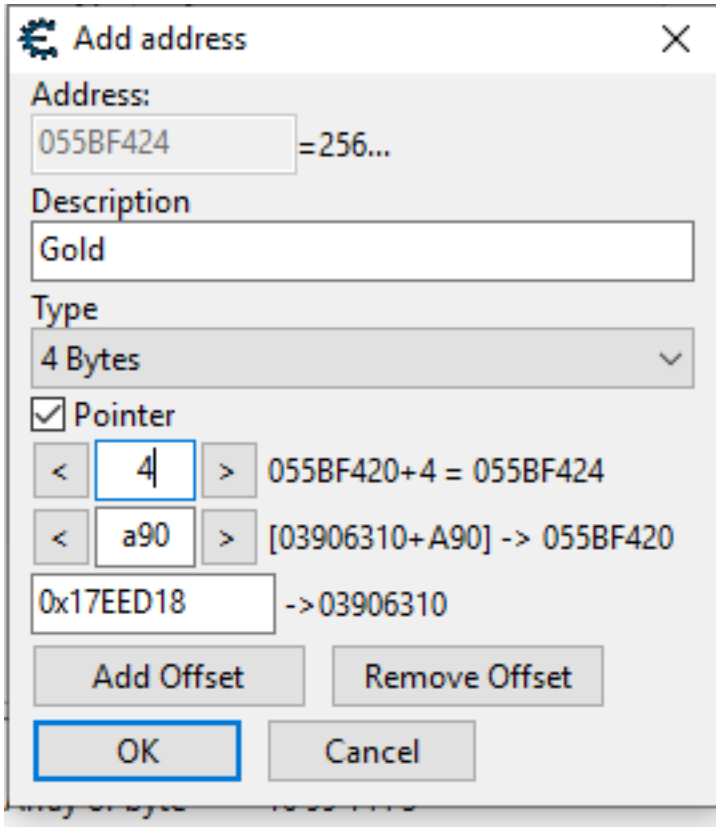
2.8.6 Alteração

Depois de encontrar o ponteiro base, agora podemos salvar seu valor para uso em projetos futuros. A partir da nossa reversão, sabemos que o valor do ouro do jogador em Wesnos é sempre: [[0x017EECB8+0x60]+0x90]+4. Para simplificar, conhecemos 0x017EECB8+ 0x60 sempre será 0x017EED18, então o deslocamento real pode ser expresso como: [[0x017EED18]+0xA90]+4.

O mecanismo de fraude nos permite adicionar manualmente um ponteiro com deslocamento como endereço. Podemos usar isso para verificar que nossos valores estão corretos. Primeiro, ligue o mecanismo de trapaça e adicione-o ao Wesnoth. Após a conexão, selecione o botão Adicionar Endereço Manual.



Na caixa pop-up, selecione a caixa de verificação do ponteiro. Ao executar isso, o mecanismo de trapaça solicitará que você insira o endereço e o deslocamento. Digite 0x017 EED18 como base e 0xA90 como deslocamento. Em seguida, adicione outro deslocamento e digite 4. Essas são todas as informações que encontramos ao dar marcha ré.



Na caixa de endereço superior, você deve notar que o mecanismo de trapaça resolveu corretamente nosso offset para a quantidade de ouro que temos atualmente. Se você desligar e reiniciar o Wesnoth, o ponteiro mudará para um novo valor para o gold.

Parte 3
Programação

3.1 Programação Fundamentos

3.1.1 Visão geral

Em capítulos anteriores, exploramos técnicas para hackear o Wesnoth, incluindo alterações de memória (como ouro) e alterações de código (como unidades de recrutamento). No entanto, todas essas mudanças só continuam até fecharmos o Wesnoth. Para recuperar esses hackers ao reabrir o jogo, precisamos repetir o processo inicial em um scanner de memória ou depurador.

Isso é chato e impossível de distribuí-lo para mais públicos. No entanto, como agora podemos vencer o DMA, sabemos que qualquer memória que precisamos mudar está sempre em uma posição estática. Portanto, podemos criar um conjunto de instruções que contenham as mudanças que queremos fazer. Criar esse conjunto de instruções de uma forma que os computadores possam entender é chamado de programação. Ao programar hackers, podemos criar programas que podem ser executados e alterar automaticamente a memória com a qual nos importamos. Também podemos distribuir esses programas para outras pessoas que querem usar nossos hackers para experimentos.

Como um todo, a programação é um tópico muito grande para ser totalmente coberto nesses capítulos. Em vez disso, vamos nos concentrar em subconjuntos de programação relacionados à criação de hackers.

3.1.2 Linguagem de programação

No Capítulo 1.1, apresentamos brevemente as linguagens de programação. Linguagens de programação permitem que o código seja escrito em um formato legível para humanos. O código é então traduzido para instruções que a CPU pode entender. Existem muitas linguagens de programação que podem ser divididas em duas categorias: como elas são executadas e como elas são executadas.

Linguagens de programação podem criar código que é executado diretamente na CPU ou por meio de um intérprete. O interpretador funciona traduzindo dinamicamente o código inicial para uma forma que a CPU possa entender. Esses dois tipos são chamados de

Linguagem de compilação e linguagem explicativa. As linguagens de programação também podem executar instruções em ordem (de cima para baixo) ou por meio de declarações e análises de funções. Esses tipos são chamados de comando e funcionais , respectivamente.

Uma linguagem pode ser classificada aplicando esses dois modificadores. Por exemplo, C é uma linguagem de comando compilada. Java é uma linguagem de comando explicativa. Haskell é um exemplo de uma linguagem funcional compilada. Linguagens de interpretação também podem ser compiladas, geralmente agrupando o intérprete às instruções iniciais.

Não existe linguagem correta ou melhor. Algumas línguas são mais adequadas para propósitos diferentes, mas todas as línguas podem atingir cada propósito. No entanto, ao programar jogos para hacker, temos várias restrições que limitam nossa escolha de idiomas. O idioma que escolhemos precisa suportar três recursos principais:

- Acesso direto à API Windows
  Modificar a memória de outros aplicativos
- Carregar e executar na CPU

Todos esses requisitos serão explicados mais tarde, mas essencialmente excluem a linguagem explicativa. Além disso, linguagens que não permitem acesso direto à memória, como Java, também são excluídas.

3.1.3 C++

Existem vários idiomas que sustentam os três critérios acima. Isso inclui C e C#, bem como versões compiladas de python. No entanto, C++ oferece uma combinação ideal de recursos de linguagem de alto nível (como classes e strings) e acesso direto à memória de baixo nível. Isso faz do C++ uma escolha ideal para hackers de jogos programados.

C++ é uma linguagem de comando compilada. Esta é uma linguagem relativamente difícil de aprender, mas precisamos entender algumas de suas características para criar hackers de jogos. Um de seus recursos mais importantes para nós é a capacidade de criar ponteiros que podem modificar diretamente o endereço de memória.

3.1.4 ponteiro

Ponteiros são outro assunto complexo que apresentaremos brevemente. Um ponteiro é uma variável que aponta para outra parte da memória. Por exemplo, o seguinte código C++:

```
int x=5; int*
y=&x;
```

Em C++, a* representa uma declaração de ponteiro. &Retorna o endereço da variável. Então, após executar esse código, a variável y aponta para a variável x. Considere os seguintes códigos:

```
*y=6;
```

Este código remove a referência (ou obtém o endereço ao qual aponta) y e atribui esse valor a 6. Após a execução deste código, a variável x também será 6, porque esse é o valor apontado por y.

Aplicando isso a hackers de jogos, suponha que encontremos um valor de ouro em 0x 12345678 e que o valor não seja alocado dinamicamente. Se quisermos carregar o programa C ++ no espaço de endereço do jogo, podemos usar um ponteiro para modificar o valor do gold:

```
int*gold=(int*)0x12345678; *
0uro = 999;
```

Após a execução, o valor de ouro em 0x12345678 agora será definido para 999 . Os ponteiros nos dão muito controle sobre a memória do jogo, mas podem ser difíceis de entender. Exploraremos isso mais a fundo nos próximos capítulos.

3.1.5 Tipos de ataques hackers

Existem três tipos principais de hackers de jogos que podem ser programados. Estes são:

- Arquivos executáveis externos
- DLL injetado (biblioteca de links dinâmicos)
- Embalagem personalizada

Cada um tem seu próprio caso de uso. Arquivos executáveis externos são procedimentos independentes que podem ser executados normalmente. Esses arquivos executáveis usam uma função integrada do Windows chamada Interface de Programação de Aplicativos (API) para ler e modificar a memória de outro arquivo executável. Em contraste, o DLL injetado precisa ser carregado na memória do jogo de alguma forma. Uma vez carregados, eles são executados na memória do jogo e podem ser acessados diretamente pela memória do jogo por meio de ponteiros . Ao criar hackers direcionados a bibliotecas de desenhos de jogos, como DirectX e OpenGL, encapsuladores personalizados são usados. Ao "embalar" os recursos originais carregando versões personalizadas dessas bibliotecas, podemos mudar a lógica de desenho do jogo.

3.2 Memória Externa Huck

3.2.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

3.2.2 Identificação

Neste capítulo, criaremos um programa C++ para modificar as moedas de ouro do jogador no Wesnoth.

3.2.3 Compreensão

No Capítulo 2.8, derrotamos o DMA no Wesnoth e posicionamos o ponteiro base do jogador em 0x017EECB8. Em seguida, determinamos o deslocamento necessário para localizar o ouro do jogador a partir do ponteiro base. Isso nos permite começar com um endereço estático, adicionar uma série de deslocamentos estáticos e, em seguida, chegar ao endereço dinâmico.

Como esses endereços e desplazamentos são estáticos, podemos criar um programa para executar essa operação. No Capítulo 3.1, apresentamos várias maneiras de atingir esse objetivo. Neste capítulo, criaremos um arquivo executável externo.

3.2.4 Estúdio Visual

Para criar um programa C++, precisamos de um compilador e um linker. O compilador é usado para converter códigos de linguagem de alto nível em códigos de operação. Em seguida, use o linker para criar arquivos executáveis que o sistema operacional entende a partir desses códigos de operação. Esses dois componentes são geralmente agrupados em um ambiente de desenvolvimento integrado (IDE). O IDE também inclui outros componentes, como conclusão de código e depuração interativa. Visual Studio é um IDE lançado pela Microsoft para Windows. A versão comunitária é gratuita

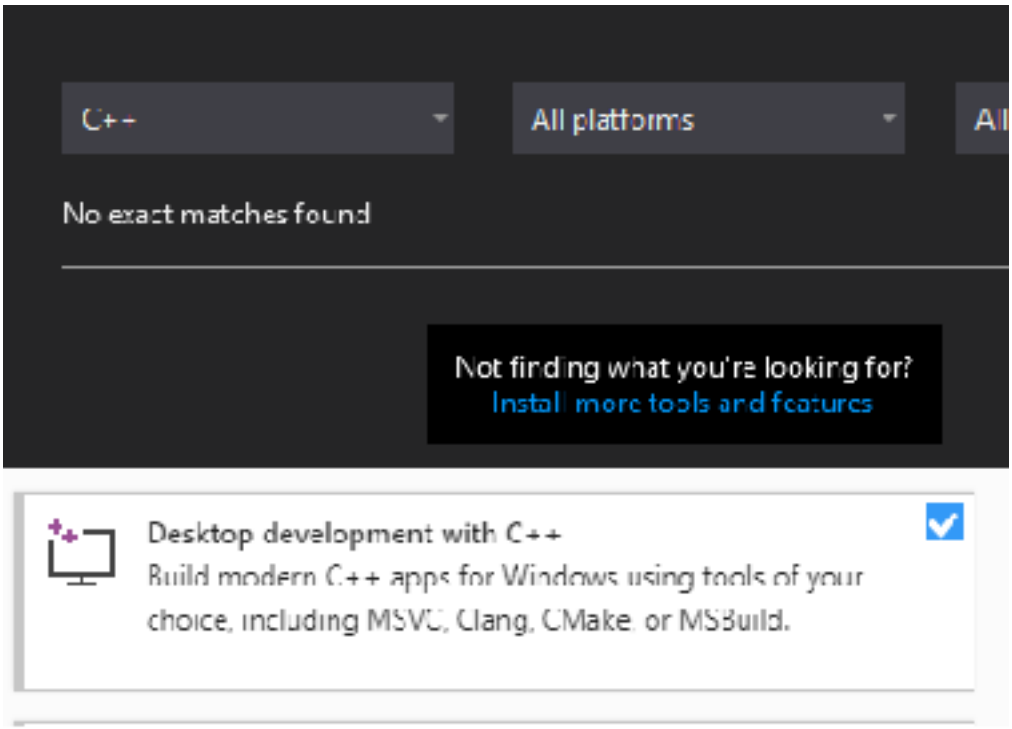
Baixe e use em seu projeto pessoal. Ele pode ser instalado usando o Chocolate, que foi instalado quando configuramos a máquina virtual pela primeira vez. Para isso, abra o prompt de comando ou Powershell e execute o seguinte comando:

```
choco instala a comunidade Visual Studio 2019
```

No Visual Studio, arquivos de código-fonte estão incluídos no projeto. Vários desses projetos podem ser incluídos em uma solução. Por exemplo, a solução Visual Studio da Wesnoth pode ser a seguinte:

```
Solução do jogo
Motor-Projeto
Placador.CPP-Código Fonte
Mestre CPP-Código Fonte UI-
Projeto
...
Projetos de rede
…
```

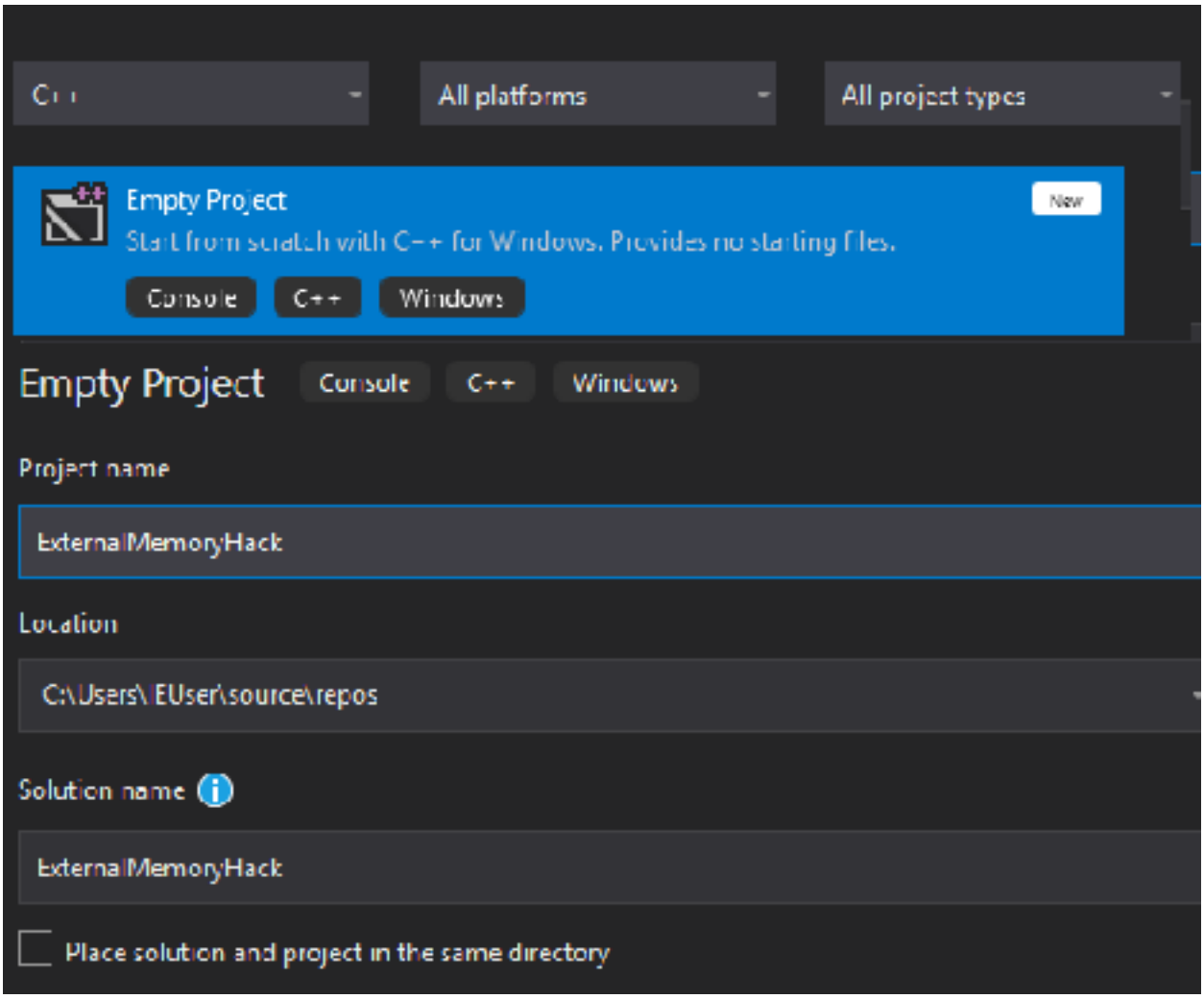
Para criar uma solução C++, primeiro precisamos instalar alguns componentes C++. Isso pode ser feito selecionando o link "Instalar mais ferramentas e recursos" e depois selecionando "Use C++ para desenvolvimento de desktop" no assistente.



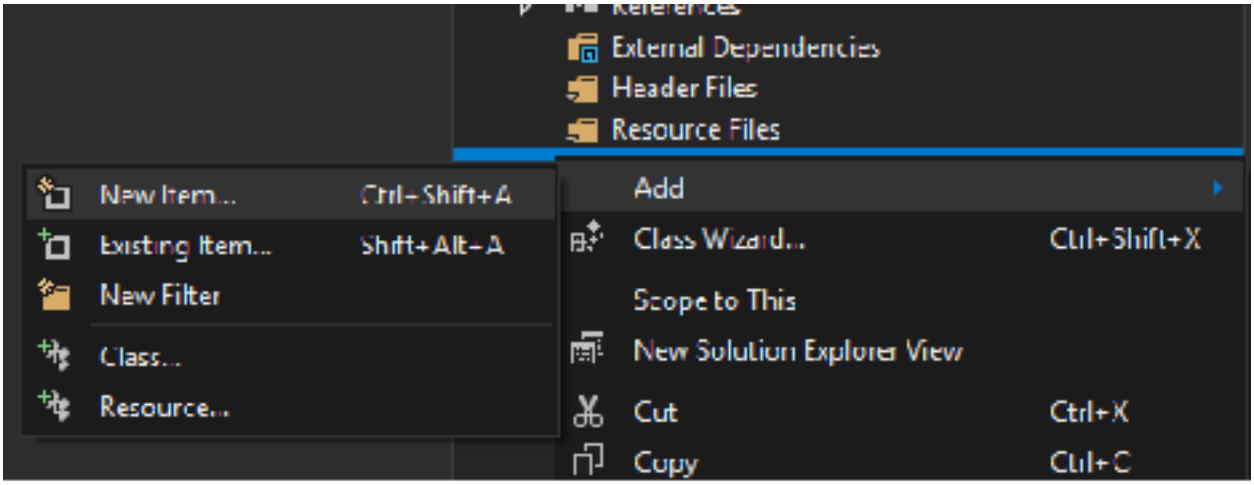
Com esses componentes instalados, agora podemos criar projetos C++ e compilá-los.

3.2.5 Criar Projetos

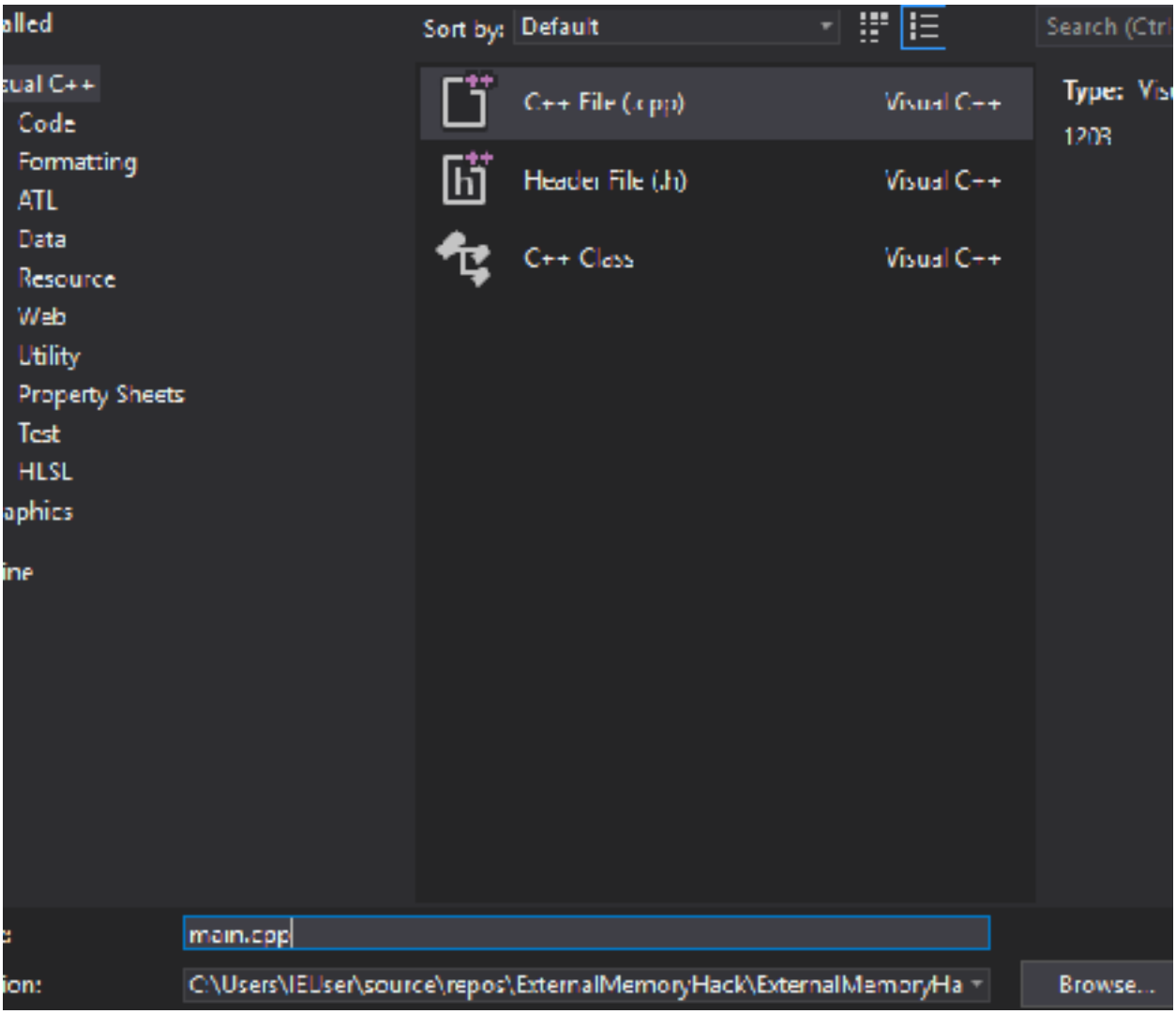
Depois que esses componentes estiverem instalados, crie um projeto C++ vazio e denomine-o Externalmemoryack.



Depois de criar um novo projeto, agora podemos adicionar arquivos de código-fonte que contêm todo o código. Para isso, clique com o botão direito no arquivo de origem e selecione Adicionar -> Novo Projeto:



No modo que aparece, selecione o arquivo C++ e o nomeie como o principal.cpp:



Depois de criar o projeto e o arquivo de origem, agora podemos começar a programar.

3.2.6 Fundamentos C++

C++ tem muitas características e é uma linguagem universal. Para nossos propósitos, vamos nos concentrar nos aspectos mais importantes da criação de hackers. Neste capítulo, precisamos aprender duas coisas sobre a linguagem:

1. O programa começa com uma função chamada main. Esta função deve retornar um valor.
2. O programa pode chamar outras funções integradas ao Windows.

Qualquer executável C++ requer uma função main que retorne números inteiros. Esta função tem dois parâmetros, que atualmente se sabe que esses dois parâmetros não são importantes. Durante a execução, o sistema operacional busca e executa a função. Quando a função retorna , ela envia um sinal ao sistema operacional de que a execução do programa foi concluída.

Além das funções que criamos, também podemos chamar outras funções integradas ao Windows. O Windows tem muitas APIs para fazer coisas como exibir texto, reproduzir som e criar arquivos. O Windows também tem uma API que nos permite ler e escrever valores durante o processo. Esses são os hackers que usaremos para nos criar.

Para usar essas funções, precisamos incluir alguns arquivos de cabeçalho. O arquivo de cabeçalho contém definições de funções definidas fora do arquivo de origem. Para ler e escrever memória, precisamos incluir o arquivo de cabeçalho Windows.h.

3.2.7 Leitura

Precisamos ler vários valores no Wesnoth para localizar nosso valor de ouro. A API que lê a memória de outro processo é chamada de ReadProcessMemory. Se você pesquisar por esse nome no Google, o primeiro resultado será um documento da Microsoft. Este documento descreve como a API funciona, incluindo quais parâmetros ela aceita e quais valores ela retorna.

Ao verificar esses documentos, podemos determinar quais valores precisamos fornecer . Para quaisquer valores que ainda precisamos, podemos determinar como obtê-los.

A função de ReadProcessMemory é definida como:

```
BOOL ReadProcessMemory (
Processe HPROSS,
LPCVOID lpbaseadres,
LPVOID lpBuffer,
SIZE_T nSize,
tamanho _T*
lpnumberofbyteesread);
```

Observando o bloco de código acima, podemos começar com o primeiro parâmetro e estudar para baixo para determinar quais valores precisamos. Primeiro, não temos um cabo de processo, então precisamos encontrá-lo. Discutiremos como fazer isso na próxima seção. Temos um endereço base (neste caso, nosso ponteiro base). O buffer precisa ser fornecido por nós, então precisamos criá-lo. O parâmetro de tamanho será o tamanho dos dados a serem lidos. Neste caso, o tamanho será de 4 bytes devido ao tamanho do registro que vemos durante a reversão. Por fim, precisamos criar outra variável para salvar o número de bytes realmente lidos quando a função é executada.

Embora isso possa parecer imparável, nos dá um ponto de partida para a programação. Primeiro, sabemos que precisamos incluir Windows.h para que possamos usar o ReadProcessMemory:

```
#include<Windows.h>
```

Em seguida, como este é um arquivo executável C++, criaremos nossa função principal:

```
int main (int argc, char**argv){
return 0;

}
```

Por fim, na função main, acima da declaração de resposta, podemos inserir uma chamada para resposta. Para valores que ainda não temos, inserimos um nome de variável. Quando descobrimos como recuperar esses valores, podemos atribuir valores a essas variáveis.

```
ReadProces_Process, 0x017EECB8, gold_value, 4, bytes_read);
```

Os valores gold_value e bytes_read são fornecidos por nós e preenchidos por APIs. Portanto , precisamos inicializar duas variáveis para salvar esses valores. Como estamos lendo 4 bytes , essas variáveis precisam ser grandes o suficiente para acomodar esse tamanho. Uma opção para fazer isso é usar um DWORD de 32 bits (ou 4 bytes) de comprimento. Precisamos colocar essas declarações acima de uma chamada para ReadProcessMemory:

```
DWORD gold_value=0;
DWORD bytes_read=0;
```

Como ambos os parâmetros devem ser ponteiros, também precisamos alterar a chamada ReadProcessMemory. Precisamos usar e passar os endereços dessas variáveis, em vez de passar os valores das variáveis:

```
ReadProces_Process, 0x017EECB8, &gold_valor, 4, &byte_read);
```

3.2.8 Iniciar o processo

Nosso próximo passo é recuperar um cabo de processo. Para isso, podemos usar uma API chamada OpenProcess. Esta API é definida como:

```
Processamento de OpenProcessos (
DWORD DWESSIDCESS, BOOL
Binherhandle, DWORD
DWESSID
);
```

A partir dessa definição, vemos que o OpenProcess retorna o cabo de um processo. Este cabo pode ser usado como o primeiro parâmetro do ReadProcessMemory. Ao visualizar o documento, queremos que o acesso que queremos seja PROCESS_ALL_ACCESS para que possamos ler e escrever o processo. O segundo parâmetro não tem nada a ver com o que estamos fazendo, então o definiremos para verdade. Precisamos encontrar o parâmetro final, então agora estamos criando uma variável. Como precisamos do resultado dessa função para chamar a ReadProcessMemory, colocaremos a chamada para ela acima da ReadProcessMemory. Nosso código agora deve ser o seguinte:

```
Mangue wesnoth_process=OpenProcess (PROCESS_ALL_ACCESS, verdade, process_id);
DWORD gold_value=0;
```

```
DWORD bytes_read=0;
ReadProces_Process, 0x017EECB8, &gold_valor, 4, &byte_read);
```

Em seguida, recuperaremos o id Process_para a chamada OpenProcess. Semelhante às duas APIs anteriores, usaremos outra API e preencheremos quaisquer informações que precisamos. Neste caso, a API será GetWindows ReadProcess. Esta API recupera o ID do processo quando um cabo de janela diferente do cabo de processo é fornecido. Esta API é definida como:

```
DWORD GetWindows ReadProssID (
HWND HWND,
LPDWORD lpdwprotessid);
```

Esta função requer um cabo de janela e uma variável para salvar o ID do processo. Como antes, adicionaremos o seguinte código acima da chamada OpenProcess:

```
DWORD Proces_ID=0;
GetWindows ReadProcess (wesnoth_Windows, &process_id);
```

Para obter um cabo de janela, podemos usar a API FindWindows. Esta função usa o nome do título da janela e retorna o cabo da janela. A definição é:

```
HWND FindWindows (
LPCSTR  lpClassName,
LPCSTR lpWindowName);
```

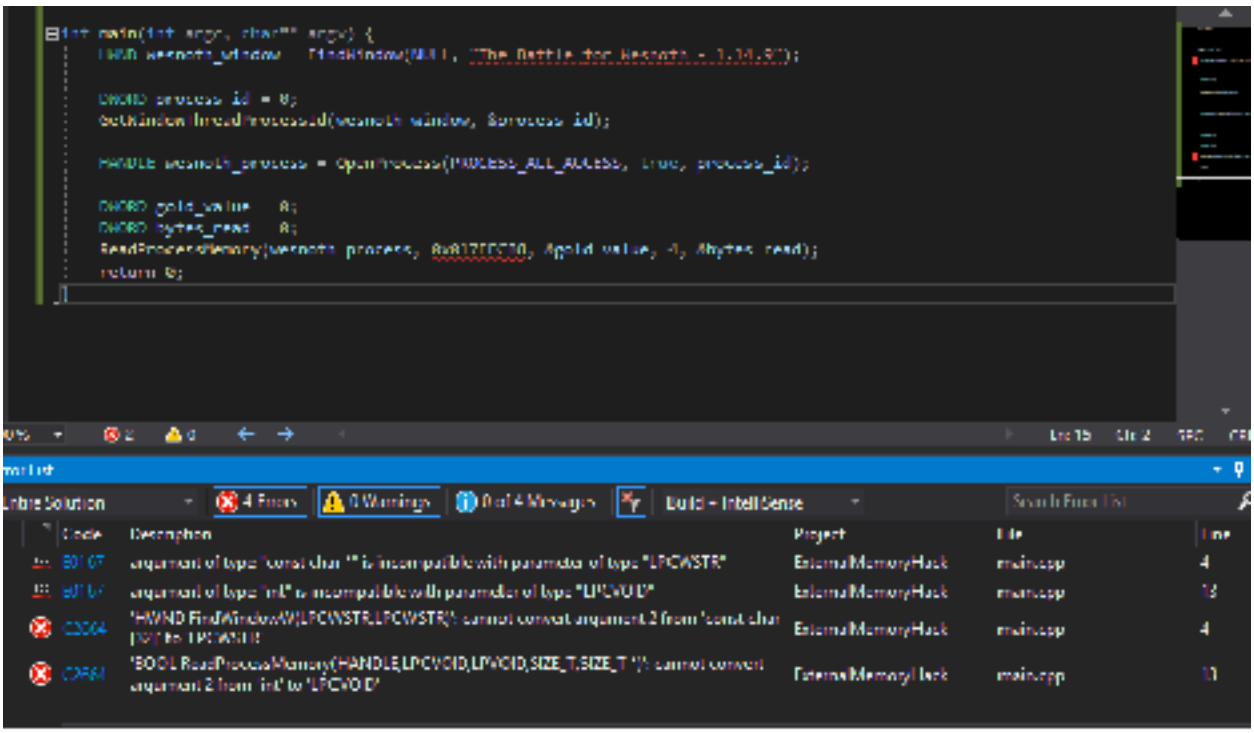
Como estamos pesquisando todas as janelas, definimos o primeiro parâmetro para NULL. Para o segundo parâmetro, conhecemos o nome da janela Wesnoth porque ela é exibida na barra de título do jogo. Podemos inserir esta chamada final na parte superior da função principal. Agora, nosso código é o seguinte:

```
#include<Windows.h>

int main(int argc, char**argv){
HWND Wesnoth_Windows=FindWindows (NULL, "Wesnoth Battle -1.14.9"); DWORD Proces
_ID=0;
```

```
GetWindows ReadProcess (wesnoth_Windows, &process_id);

Mangue wesnoth_process=OpenProcess (PROCESS_ALL_ACCESS, verdade, process_id);

DWORD gold_value=0;
DWORD bytes_read=0;
ReadProces_Process, 0x017EECB8, &gold_valor, 4, &byte_read);

Retorne a 0
;}
```

3.2.9 Parâmetros de fundição

O Visual Studio exibirá alguns erros para o código que escrevemos. Isso ocorre porque não convertemos as duas variáveis corretamente. Portanto, o compilador não consegue entender como queremos passar dados para a função. Se quisermos compilar nosso programa, precisamos corrigir esses erros. Felizmente, ao ler a lista de erros no Visual Studio, podemos determinar o que precisamos fazer para corrigi-los.



Vamos resolver dois erros de reconhecimento que ocorreram na linha 13 do código. O primeiro erro na linha 13 é que os parâmetros do tipo "int" são compatíveis com os tipos de parâmetros.

"LPCVOID". Isso mostra que temos um parâmetro, que deve ser um número inteiro LPCVOID. O segundo erro não conseguiu converter o parâmetro 2 de "int" para "LPCVOID" indica qual parâmetro é esse. O parâmetro 2 é nosso endereço 0x017EECB 8. Se usarmos o Google LPCVOID, o primeiro resultado é a documentação da Microsoft sobre o LPCVOID. O documento nos mostra que LPCVOID é definido como void:*

```
typedef const void*LPCVOID;
```

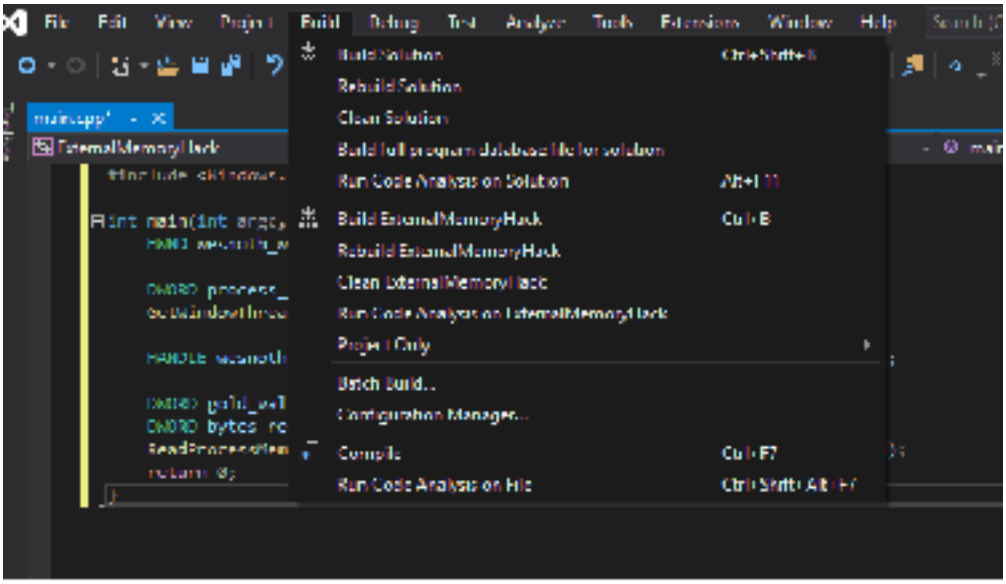
Para resolver esses erros, podemos converter o endereço para void*. O código gerado é o seguinte:

```
ReadProces_Process, (void*) 0x017EECB8, &gold_valor, 4, &byte_read);
```

Com isso, esses erros desaparecerão. Agora podemos verificar o segundo conjunto de erros que ocorreu na linha 4 sobre chamadas do FindWindows. Esses erros indicam que a string Wesnoth Battle-1.14.9 não foi lançada corretamente. É um char* constante que precisa ser convertido em LPCWSTR. Para isso, podemos usar o prefixo da string L. Nossa chamada Findwindows agora parece:

```
HWND Wesnoth_Windows=FindWindows (NULL, L "The Battle for Wesnoth-1.14.9");
```

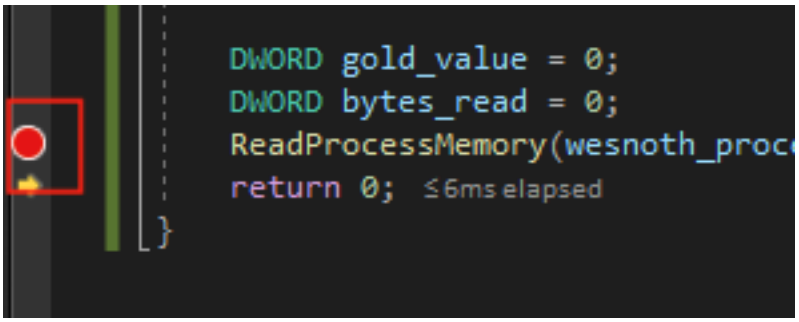
Ao fazer essa alteração, o erro desaparecerá e agora podemos compilar e executar nosso programa. Para isso, vá para o item de menu "Gerar" e selecione "Gerar Soluções". Uma vez concluído, teremos um programa executável.



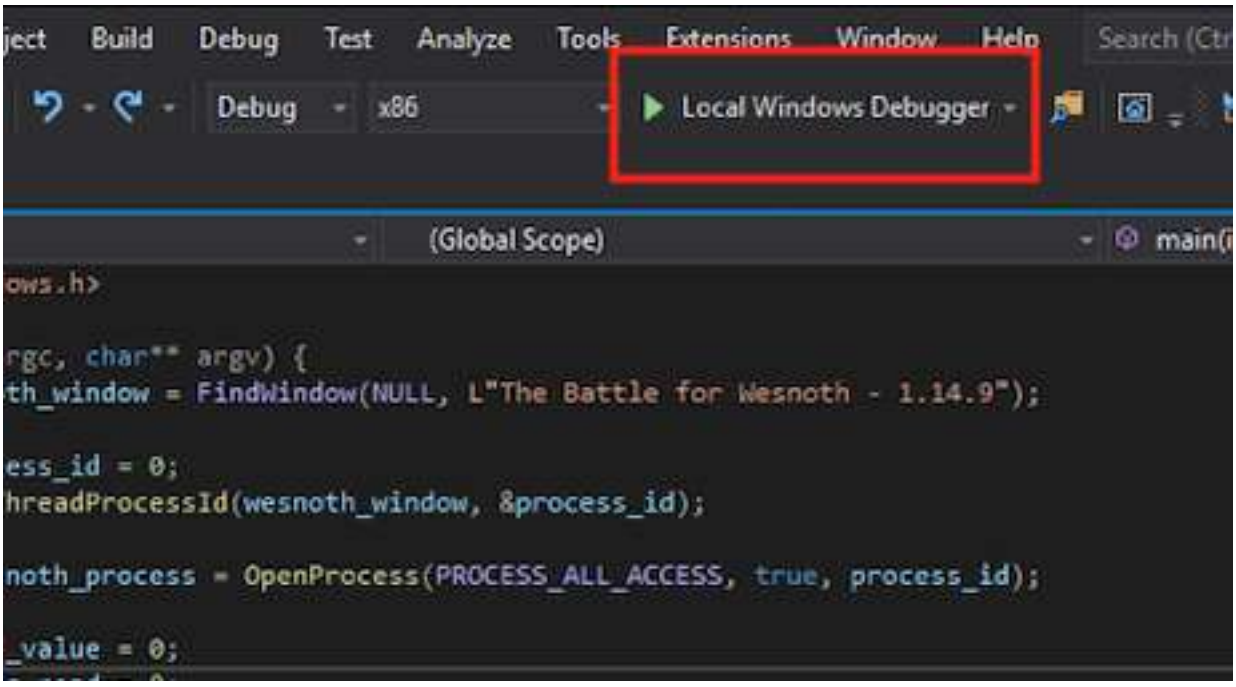
3.2.10 Depuração

Para verificar se esse código é válido, precisamos garantir que estamos lendo o valor correto no endereço de memória 0x017EECB8. Para isso, depuraremos nosso programa no Visual Studio e compararemos os resultados da chamada ReadProcessMemory com o mecanismo de trapaça.

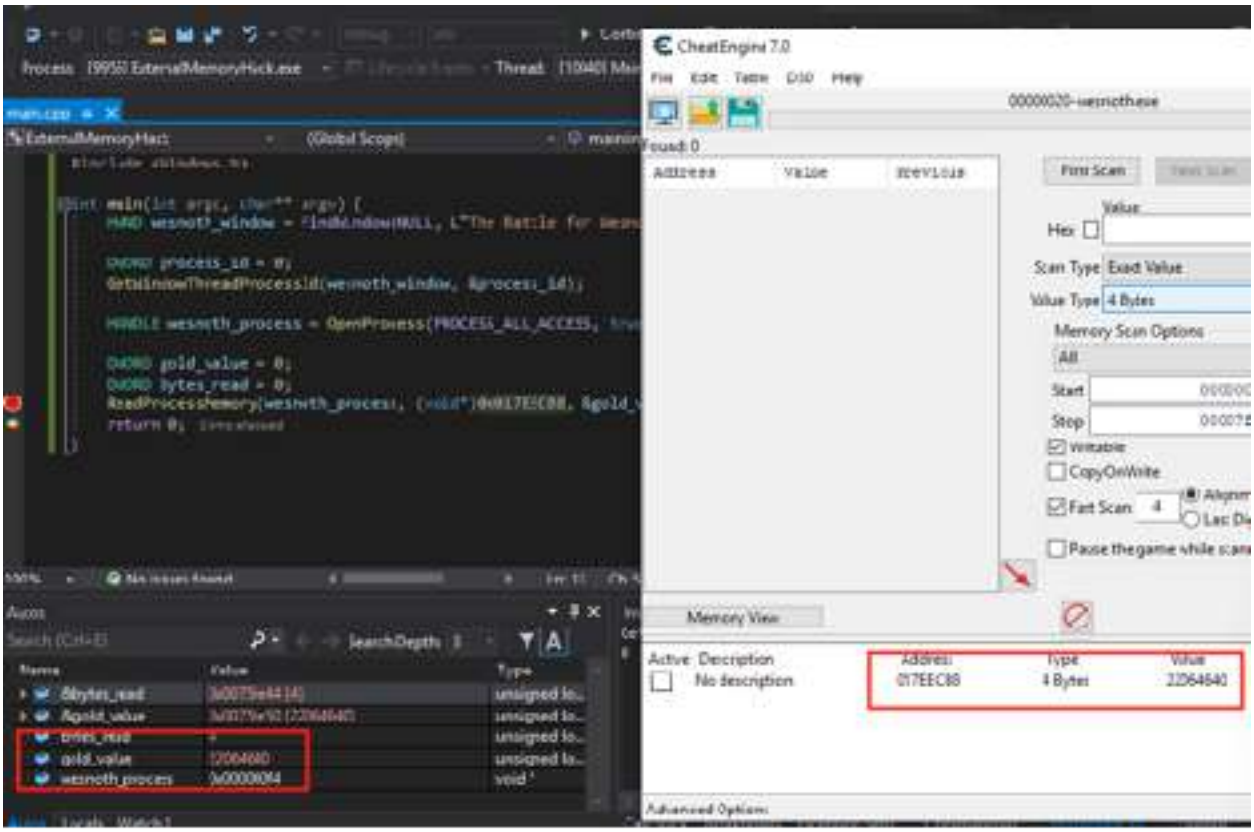
Primeiro, ligue o mecanismo de trapaça e adicione manualmente o endereço 0x017EECB 8. Em seguida, defina um ponto de interrupção na linha de readcheck-memory. Isso pode ser feito clicando com o botão esquerdo na área à esquerda da linha de código onde você deseja interromper. Se operado corretamente, um círculo vermelho aparecerá.



Em seguida, clique no botão local Windows depurador no topo do IDE. Isso começará a executar um programa com um depurador.



Como já escrevemos o código-fonte, depurar nossos programas será muito mais fácil do que depurar o Wesnoth. Quando chegamos ao ponto de interrupção, o depurador aparece, permitindo-nos explorar vários elementos do código, incluindo variáveis. Essas variáveis serão exibidas no canto inferior esquerdo do IDE. Para garantir que estejamos lendo a memória corretamente, queremos visualizar a variável gold_value e garantir que seu valor corresponda ao mecanismo de trapaça.



Devido à correspondência de valores, sabemos que estamos lendo corretamente a memória no Wesnoth. Clique no botão Continue na parte superior do IDE para concluir a execução do programa. Agora podemos continuar buscando o valor do nosso ouro.

3.2.11 DMA

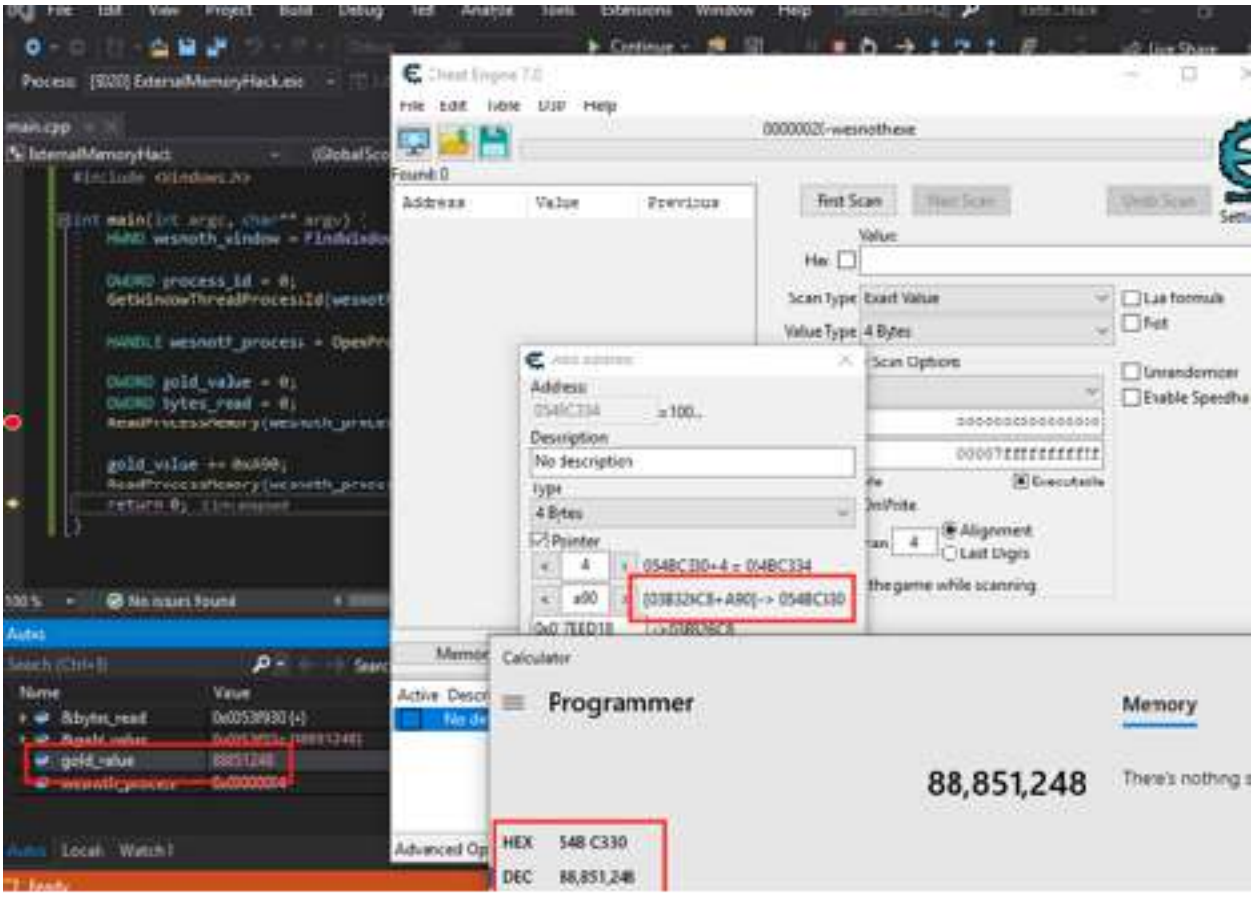
Agora que podemos ler a memória, agora podemos recuperar o valor do nosso ouro. No Capítulo 2.8 , confirmamos que nosso endereço dourado é armazenado em [[0x017EECB8+0x60]+0xA90]+4. Isso pode ser ainda mais simplificado para [[0x017EED18]+0xA90]+4. Para recuperar o endereço dourado em nosso programa, podemos primeiro ler o valor em 0x017EED18 e depois adicionar 0xA90 para

Esse valor. Podemos então ler o endereço e adicionar 4 a ele. Quando fizermos isso, teremos nosso endereço dourado.

Para concluir todo o processo, podemos usar a mesma chamada ReadProcessMemory que o código anterior. Primeiro, leremos o valor [[0x017EED18]+0xA90].

```
DWORD gold_value=0;
DWORD bytes_read=0;
ReadProces_Process, (void*) 0x017EED18, &gold_value, 4, &bytes_read);


gold_value+=0xA90;
ReadProces_Process, (void*) gold_value, &gold_value, 4, &bytes_read);
```

Podemos usar o mecanismo de fraude para verificar o deslocamento para garantir que lê o valor corretamente. Podemos então usar pontos de interrupção para garantir que os valores correspondam na segunda chamada de ReadProcessMemory. Como o Visual Studio exibe variáveis em formato decimal, precisamos converter esses números em hexadecimal para verificação.



Como nossos valores correspondem, podemos adicionar um deslocamento final 4 ao endereço para recuperar nosso endereço dourado. Em seguida, vamos nos concentrar em escrever memórias.

```
gold_value+=4;
```

3.2.12 Memória de gravação

A API que grava outra memória de processo é chamada de WriteProcessMemory . Sua definição é muito semelhante à da ReadProcessMemory:

```
BOOL WriteProcessMemory (
Processe HPROSS,
LPVOID lpbaseadres,
Buffer lpcvoid lp,
SIZE_T nSize,
SIZE_T*lPNUMBEOFBYTESWRITTEN);
```

A principal diferença é que esta função escreve o valor do buffer para parte da memória em vez de ler parte da memória no buffer. Como antes, precisamos declarar duas variáveis para o buffer e o número de bytes escritos.

```
DWORD new_gold_value=555;
DWORD BYTE_WRTEN=0;
```

Podemos então chamar o WriteProcessMori quase da mesma maneira que o ReadProcessMori. Assim como ReadProcessMemory, convertemos gold_value para (void*):

```
WriteProcess_Process_Process_Process_Process_Process_Process_Process_
Process_Process_Process_Process_Process_Process_, (void*)
```

Quando isso for executado, nosso ouro será definido para 555 e nossos hackers o completarão . Agora podemos executar esse arquivo executável sempre que quisermos mudar nosso ouro. Também podemos distribuí-lo para outros jogadores para execução em suas máquinas.

Os códigos completos deste capítulo podem ser comparados no Apêndice A.

3.3 Memória DLL Huck

3.3.1 Objetivos

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

3.3.2 Identificação

Neste capítulo, criaremos uma biblioteca de links dinâmicos (DLL) para modificar o ouro do jogador no Wesnoth. Este DLL modifica as moedas de ouro do jogador toda vez que o usuário pressiona um botão.

3.3.3 Compreensão

No capítulo anterior, criamos um programa externo C++ que usa ReadProcessMemory e WriteProcessMemory para modificar o ouro do jogador. Embora essas APIs sejam úteis, elas também têm algumas limitações. Devido à sua definição, eles exigem que convertemos parâmetros em tipos definidos. Suas definições facilitam a modificação de valores simples como o gold, mas dificultam a leitura e a escrita de classes completas ou tipos de dados complexos.

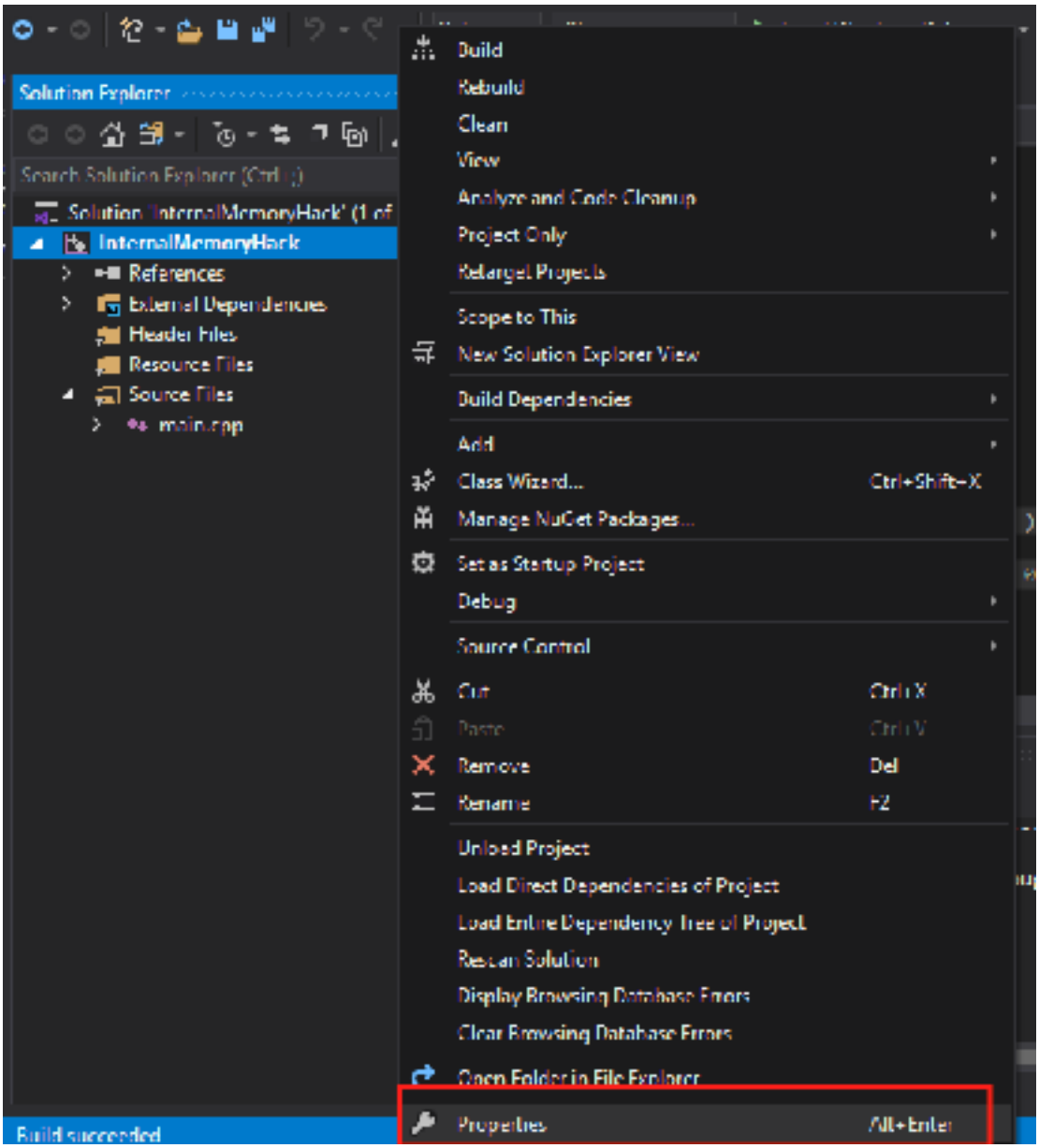
Como essas APIs são executadas a partir de programas externos, é difícil para nós fazer algumas coisas, como monitorar botões no jogo. Além disso, se quisermos criar uma caverna de código no jogo, temos que converter manualmente essa caverna de código para sua representação de código de operação. Depois, precisamos encontrar um local de memória para colocá-lo para que o WriteProcessMemory possa funcionar.

Para contornar todas essas limitações, podemos injetar um DLL na Wesnoth. Uma vez injetado, este DLL será carregado no jogo e a memória do jogo pode ser acessada diretamente usando ponteiros. Também podemos criar threads que são executados dentro do jogo, permitindo-nos monitorar entradas do usuário e outros eventos.

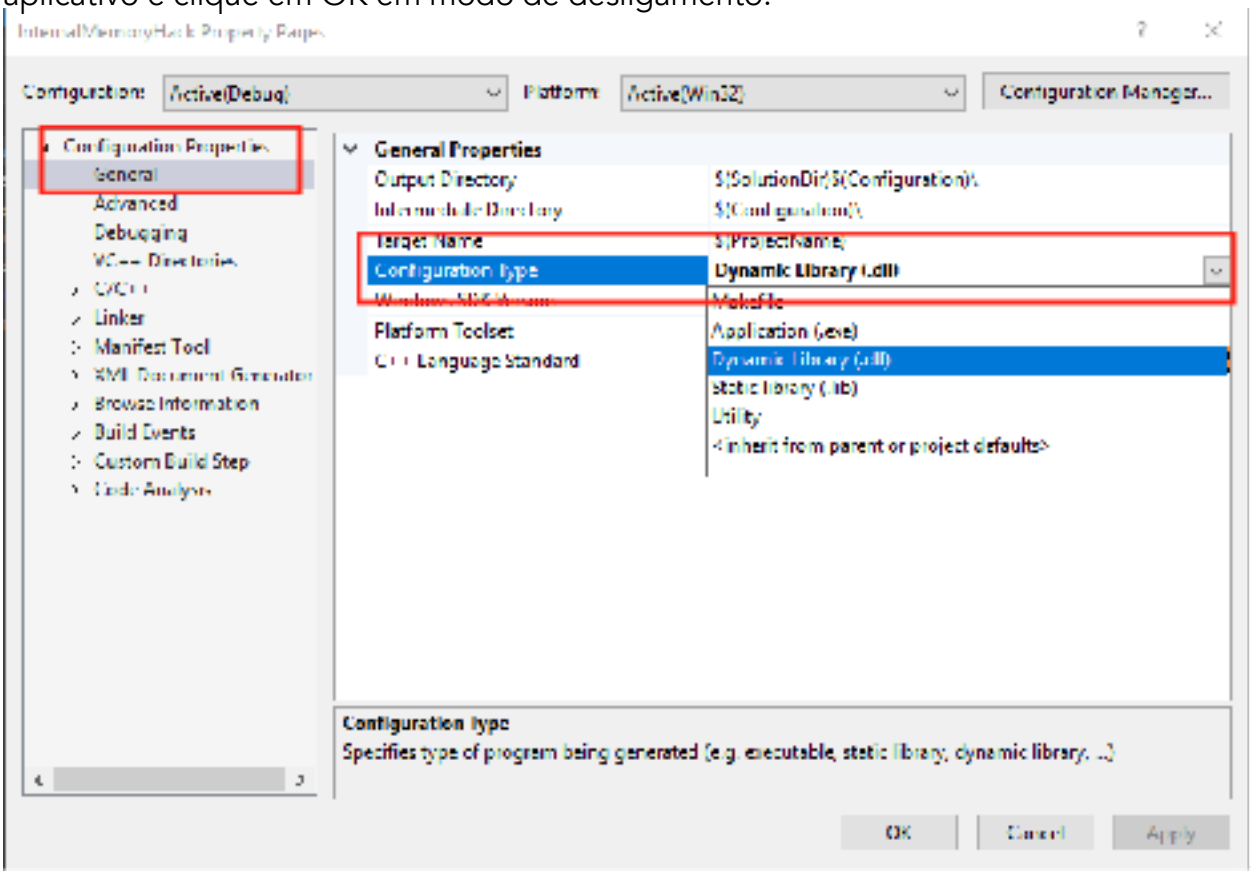
3.3.4 Criar DLL

Primeiro, crie um projeto vazio, desta vez chamado Internalmemoryack. Após criar o projeto, adicione um arquivo main.cp. O processo de execução dessas etapas é o mesmo do capítulo anterior.

Por padrão, itens vazios no Visual Studio são configurados para ser construídos como arquivos executáveis. Para construir um DLL, precisamos alterar o tipo de configuração do projeto. Isso pode ser feito nas preferências do projeto. Primeiro, clique com o botão direito no nome do projeto e selecione o item de menu Propostas.



Em seguida, selecione o Normal sob as propriedades de configuração. Em seguida, mude o tipo de configuração do aplicativo para a biblioteca dinâmica. Selecione o aplicativo e clique em OK em modo de desligamento.



Nossos projetos agora serão construídos como DLL e não como arquivos executáveis.

3.3.5 Conhecimento básico de DLL

DLL não pode ser executado por conta própria. Em vez disso, eles precisam ser carregados em arquivos executáveis. DLL permite que os desenvolvedores criem bibliotecas de funções que podem ser carregadas dinamicamente. Essas bibliotecas podem então ser usadas em vários arquivos executáveis e reduzir a quantidade de código que os desenvolvedores precisam escrever.

Por exemplo, o user32.dll contém código que exibe modos, alarmes e outros elementos da UI do Windows. A maioria dos arquivos executáveis publicados para Windows carrega automaticamente esse DLL e não precisa do código original para acessá-lo. Se a Microsoft atualizar este código e alterar a aparência da caixa de alarme, todos os arquivos executáveis que carregam esta biblioteca se beneficiarão dessa alteração.

Há várias diferenças entre DLL e arquivos executáveis comuns. Para nossos propósitos, precisamos entender três deles:

1. O DLL tem uma função DllMain em vez de uma função main.
2. Quando um processo carregar ou desinstalar um DLL, esta função DLMAIN é chamada.
3. O DLL é executado em seu processo pai. Variáveis declaradas em DLL são criadas na memória pai.

A função DllMain tem parâmetros diferentes da função main. É definido como:

```
Boolean WINAPI DllMain(
_in_Hinsttdll, _in_dword
fdweason, _in_lpvoid
lpvreservid);
```

O parâmetro fdwReason contém o motivo para chamar a função DllMain. Por exemplo, quando um DLL é carregado em um processo, o parâmetro permanece no valor de 1. Esse valor também é definido pela constante DL_Proces_Attach. Para garantir que nosso código seja executado apenas uma vez, verificaremos esse parâmetro no hack final.

Como os DLLs são executados na memória de outro processo, precisamos carregá-los de alguma forma. Em ataques de hackers, isso é frequentemente chamado de injeção porque carregamos erroneamente DLL no processo. Geralmente é difícil detectar se o DLL foi injetado com sucesso. Uma abordagem é adicionar um depurador ao processo e observar todos os módulos de carga do processo. No entanto, essa abordagem pode ser demorada e nem sempre viável. Outra maneira é criar um DLL que exiba um indicador óbvio quando injetado. É assim que usaremos para testar a injeção de DLL.

3.3.6 Caixa de mensagens

A API Windows tem a função de exibir caixas de mensagens no processo. Esta função é definida como:

```
Caixa de mensagens int (
HWND HWND, LPCTSTR
lpText, LPCTSTR
LPCPTION, UINT uType

);
```

Entretanto, devido à maneira como C++ processa conversões de parâmetros, podemos ignorar os tipos desses valores. Ao chamar a função MessageBox como mostrado abaixo , exibiremos uma caixa de mensagens em branco com um título incorreto e sem texto.

```
Caixa de mensagens (0,0,0,0);
```

Podemos usar esse comportamento para garantir que nosso DLL seja injetado com sucesso no Wesnoth. No principal .cp, adicione o seguinte código:

```
#include<Windows.h>

BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid
){
Caixa de mensagens (0,0,0
,0); Retornar a verdade;

}
```

Sempre que um DLL é carregado ou desinstalado, esse código fará com que nosso DLL exiba uma caixa de mensagens no processo pai. Usaremos esse comportamento para garantir que nosso DLL seja injetado com sucesso. Crie este código usando a opção Build para gerar DLL. Este DLL será colocado no local especificado quando você criou o projeto. Por padrão, em nossas máquinas de laboratório, isso seria C:\ Users\IEUser\source\repos\Internalmemoryack\Debug\Internalmemoryack.dll.

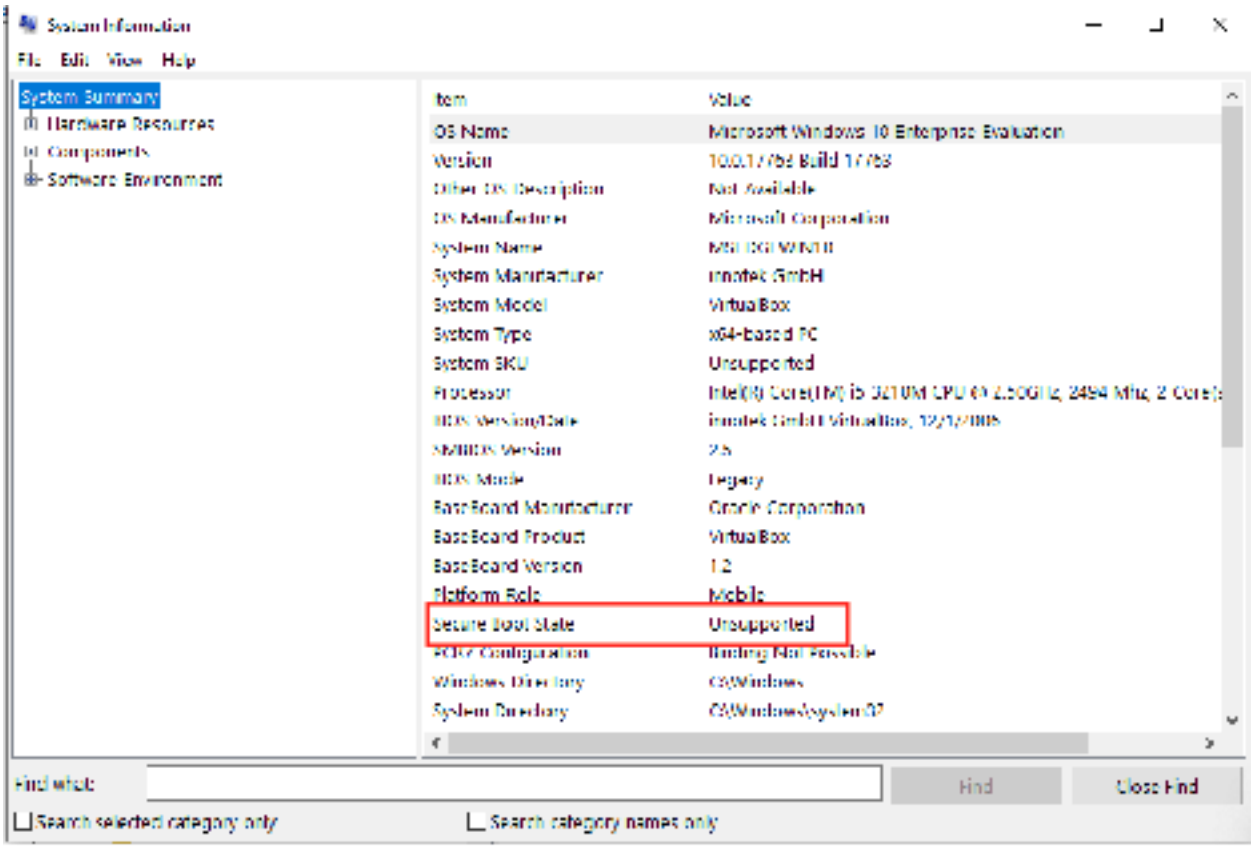
3.3.7 Injeção de DLL

Os DLs geralmente são carregados em processos usando a API Loadlibirary. Entretanto, como não modificamos o código-fonte original do jogo, precisamos encontrar outra maneira de carregar nosso DLL no Wesnoth.

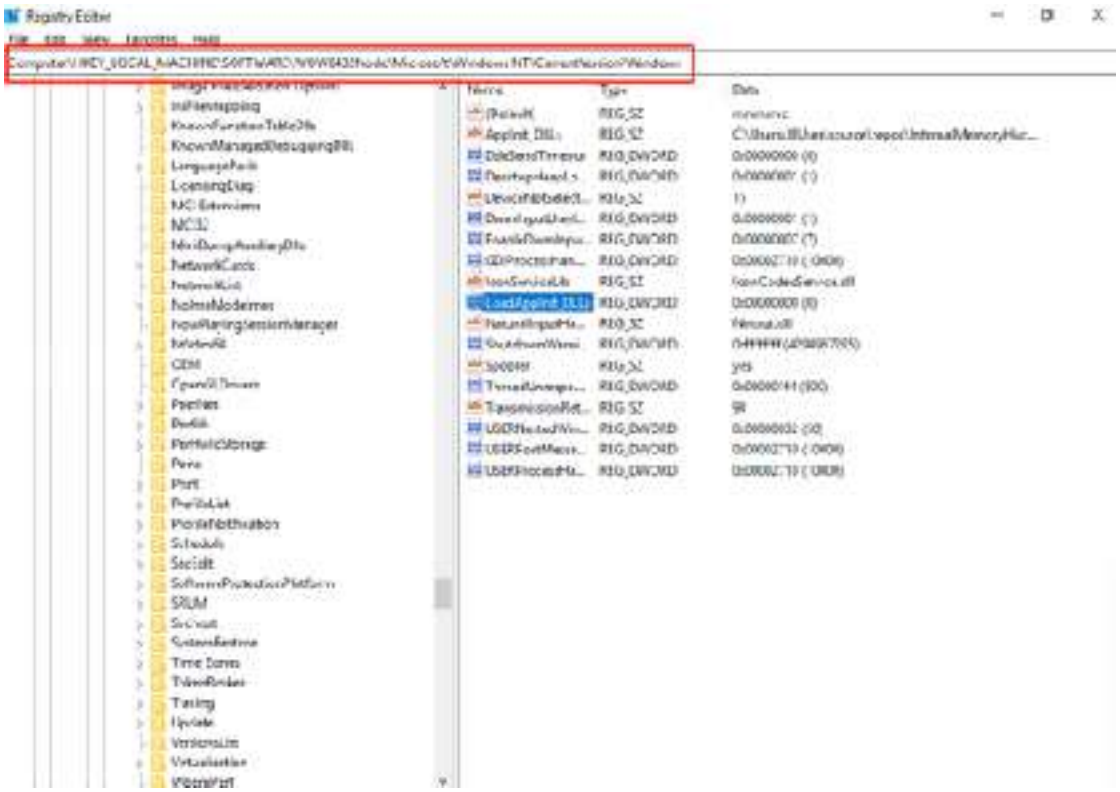
Um método que podemos usar é o injetor DLL. Um injetor DLL é um programa externo que cria uma thread dentro do processo de destino. Isso é feito usando a API CreaterEMOTETHEAD. O thread então chama a API Loadlibarry dentro do processo. No Capítulo 7.1, explicaremos como criar um injetor DLL.

Neste capítulo, usaremos um recurso do Windows que injeta DLL definido pelo usuário em cada arquivo executável iniciado. Este recurso é chamado de AppInit_DLLs e pode ser controlado por meio de um registro.

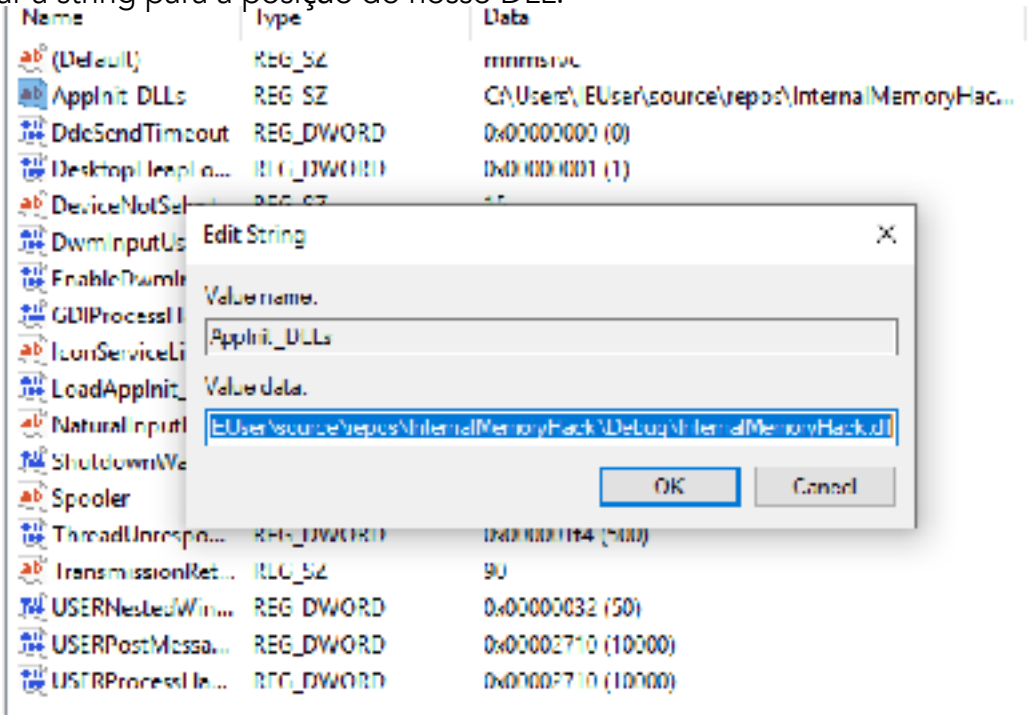
Como esse recurso é frequentemente usado por malware, o Windows 10 precisa desativar a inicialização segura para que ele funcione. Por padrão, o VirtualBox não suporta esse recurso e será desativado. Se você estiver usando hardware real, precisará desativá-lo pelo BIOS. Seu status atual pode ser determinado por meio do programa de informações do sistema:



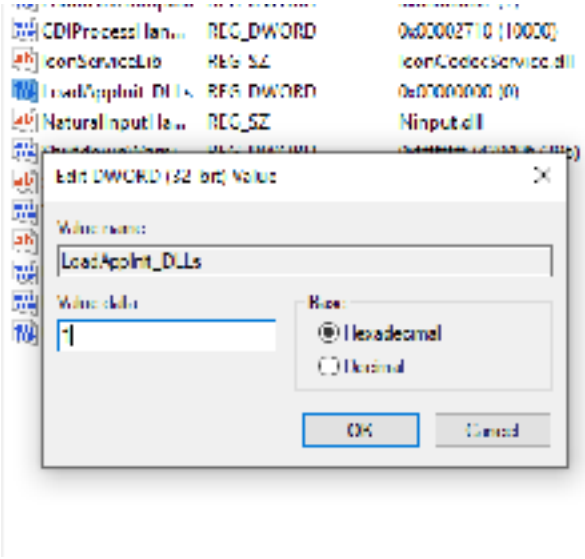
Depois que o boot de segurança for desativado, precisamos modificar o registro para habilitar AppInit_DLLs. Isso pode ser feito abrindo primeiro o programa regedit. O Registro Windows contém teclas e valores que alteram a funcionalidade do sistema operacional e do programa individual. É semelhante ao sistema de arquivos do Windows porque essas teclas e valores estão contidos no caminho. O caminho para a função AppInit_DLL em computadores Windows de 64 bits é Computer\HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Windows\Windows\Windows\CurrentVersion\Windows. Navegue para este local em regedit.



AppInit_DLLs carregará qualquer DLL especificado no valor AppInit_DLLs em todos os programas iniciados. Clique duas vezes no valor AppInit_DLLs para alterar a string para a posição do nosso DLL:



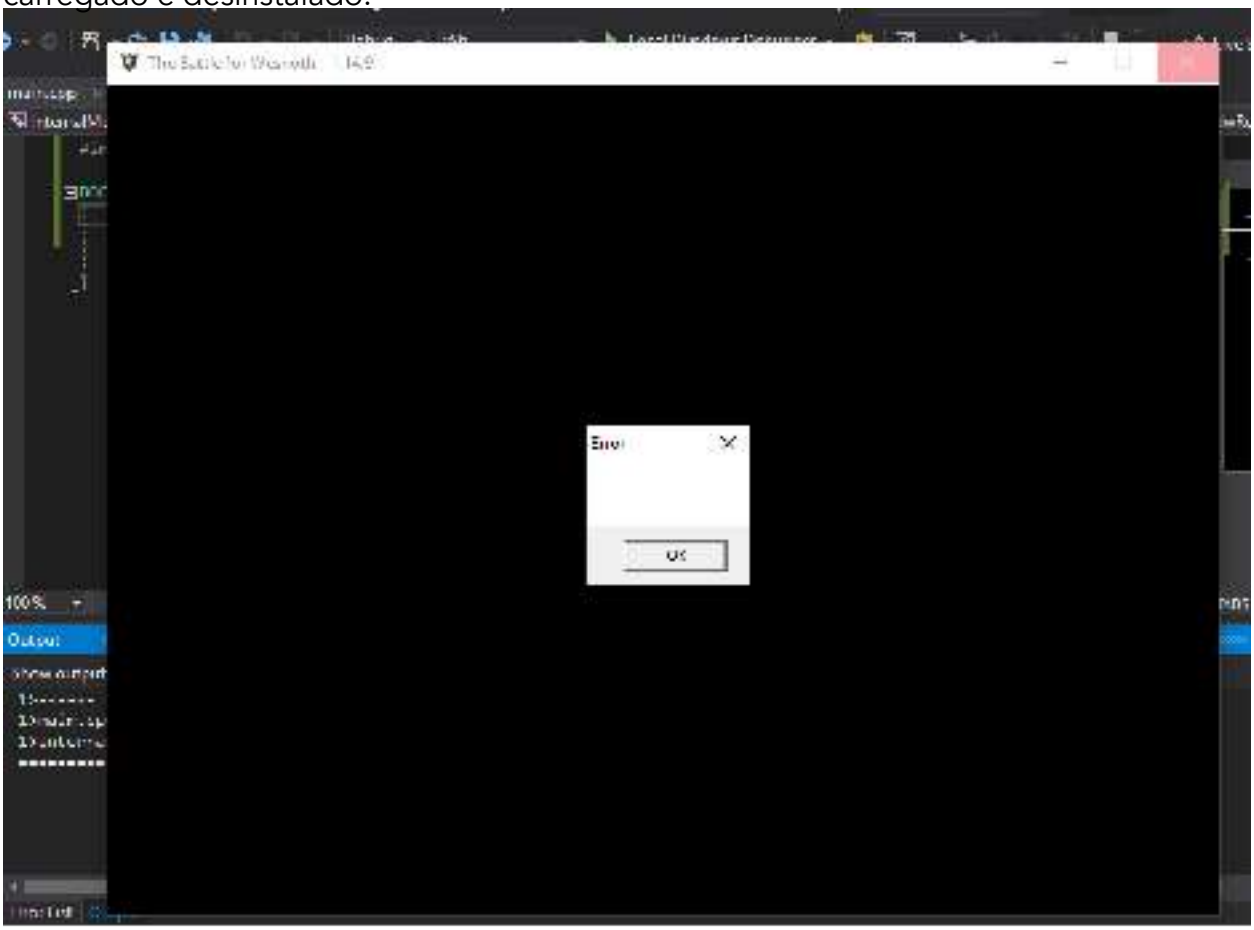
Em seguida, precisamos habilitar esse recurso alterando o valor do LoadAppInit_DLLs. Após fazer essa mudança, nosso DLL será carregado em cada novo processo. Quando esse valor é definido para 1, esse recurso é ativado. Quando estiver 0, esse recurso será desativado.



Após essas alterações, a chave do registro deve ser a seguinte:

Name	Type	Data
(Default)	REG_SZ	mnmsrvc
AppInit_DLLs	REG_SZ	C:\Users\IEUser\source\repos\ nternalMemoryHac...
DdeSendTimeout	REG_DWORD	0x00000000 (0)
DesktopHeapLo...	REG_DWORD	0x00000001 (1)
DeviceNotSelect...	REG_SZ	15
DwmInputUsesI...	REG_DWORD	0x00000001 (1)
EnableDwmInpu...	REG_DWORD	0x00000007 (7)
GDIProcessHan...	REG_DWORD	0x00002710 (10000)
IconServiceLib	REG_SZ	IconCodecService.dll
LoadAppInit_DLLs	REG_DWORD	0x00000001 (1)
NaturalInputHa...	REG_SZ	Ninput.dll
ShutdownWarni...	REG_DWORD	0xffffffff (4294967295)
Spooler	REG_SZ	yes
ThreadUnrespo...	REG_DWORD	0x000001f4 (500)
TransmissionRet...	REG_SZ	90
USERNestedWin...	REG_DWORD	0x00000032 (50)
USERPostMessa...	REG_DWORD	0x00002710 (10000)
USERProcessHa...	REG_DWORD	0x00002710 (10000)

Agora podemos começar Vesnos. Ao iniciar o jogo, várias caixas de mensagens devem aparecer indicando que nosso DLL foi injetado com sucesso e está sendo carregado e desinstalado.



Uma coisa importante a lembrar é que o AppInit_DLL injetará DLL em cada processo iniciado. Isso inclui o processo de geração de DLL quando fazemos alterações. Para evitar quaisquer problemas, precisamos desativar esse recurso ao construir DLL. Após testar seu DLL, certifique-se de definir o valor do loadpinit_DLLs para 0. Após construir um DLL, defina o valor de volta a 1 para reativar a injeção de DLL.

3.3.8 Criar threads

Agora que verificamos que a injeção de DLL está funcionando, podemos começar a programar nossos hackers. Esperamos que este DLL espere que o usuário clique no botão antes de alterar o ouro. Para isso, criaremos um thread no processo Wesnoth. Este thread continuará em execução até que o jogo saia.

Primeiro, alteraremos nosso DLMAIN para que ele execute nosso código somente quando nosso DLL for carregado no processo pela primeira vez. Isso garantirá que criemos apenas um thread no jogo. Podemos fazer isso verificando os parâmetros fdwReason:

```
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid
){
Se(fdwReason==DL_Process_Attach){
//Código a ser executado ao carregar o processo}


Retornar a
verdade;}
```

Para criar threads em um processo, podemos usar a API CreatHRead. É definido como:

```
CreatHEAD com o cabo (
lpsecurit_atributos lpthreadatos, SIZE_T
dwstack SIZE,
LPTHREAD_START_ROUTINE lpstartadres, _Drv_
aliassemem LPVOID lpParameter,
DWORD DWCREAGS, LPDWORD LPTHREAID

);
```

Como criamos uma thread sem propriedades especiais no Wesnoth, podemos ignorar a maioria desses parâmetros. O único parâmetro com o qual nos importamos é lpstartadress, que representa a função que queremos executar quando o thread é iniciado. Como essa função não requer retorno, nós a criamos como uma função void.

```
void injectid_THREAD(){

}

BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

Se(fdwReason==DL_Process_Attach){
Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL, 0, NULL);

}

Retornar a verdade;
```

```
}
```

Após o carregamento, o código cria um thread que executa a função injected_thread e depois sai. Para garantir que o thread permaneça ativo, usaremos o loop while infinito na função injected_thread:

```
while(TRE){
Sleep(1);

}
```

O loop while será executado até que sua condição seja false. Como o verdadeiro nunca pode ser igual ao verdadeiro, esse loop de while continuará em execução até que nosso thread saia devido ao fechamento do jogo. Para evitar que nossos threads diminuam a velocidade, podemos usar a API do Sono para pausar sua execução por um milissegundo.

Botão de detecção 3.3.9

Para detectar teclas, podemos usar a API GetasyState. Isso requer um parâmetro, que é a chave para verificar. Se a tecla estiver fechada, ela retornará a verdade. Caso contrário, retornará ao false. Para este capítulo, verificaremos se o usuário pressiona M:

```
while(true){if(GetasyNCKEYTE ('
M')) {//Alterar moedas de ouro do
jogador}


Sono (1);}
```

Um aviso importante sobre o GetAsyncKeyState é que, se a tecla for pressionada e segurada , ela continuará retornando a verdade. Isso não afetará o que temos neste capítulo, mas se quisermos desativar e ativar um valor no futuro, precisamos considerar esse comportamento.

3.3.10 ponteiro

No Capítulo 3.1, discutimos ponteiros. Como nosso DLL é injetado no Wesnoth, podemos acessar a memória do jogo usando ponteiros. Isso nos permite contornar

ReadProcessMemory e WriteProcessMemory. No entanto, ainda usaremos os mesmos deslocamentos e endereços usados no capítulo anterior.

Primeiro, obteremos o endereço do player lendo o valor em 0x017EED18:

```
DWORD*player_base=(DWORD*)0x017EED18;
```

Isso declarará player_base como um ponteiro apontando para o valor DWORD. Ele apontará para o local do nosso jogador 0x017EED18. Podemos então cancelar a referência deste ponteiro para "ler" ou recuperar o valor. Usando isso, podemos obter o endereço do jogo adicionando desplazamento:

```
DWORD*tame_base=(DWORD*)(*player_base+0xA90);
```

Por fim, podemos cancelar a referência ao endereço game_base e adicionar um deslocamento para recuperar nosso valor gold. Em seguida, podemos remover a referência a este valor de gold e definir seu valor diretamente:

```
DWORD*Good=(DWORD*)(*gami_base+4); *
Ouro = 999;
```

Depois de construir o DLL e reativar o loadpinit_DLLS, podemos injetar esse hack no Wesnoth. Crie um jogo e pressione a tecla "M". Quando você mover sua câmera, o valor dourado será atualizado para nosso novo valor. O código de comparação completo está disponível no Apêndice A.

3.4 Código Cave e DLL

3.4.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

3.4.2 Identificação

Nosso objetivo neste capítulo é criar uma caverna de código em DLL. Sempre que selecionamos uma descrição do terreno, a caverna de código é executada. Antes de chamar a caixa de descrição do terreno, a Code Cave nos dará 888 moedas de ouro.

3.4.3 Compreensão

No Capítulo 2.6, criamos uma caverna de código na memória do jogo. Em seguida, ajustamos o código de operação na função de descrição do terreno para jmp para esta caverna de código. Usamos o montador de instruções integrado do x64dbg para criar cavernas de código e ajustar esses códigos de operação.

Para criar esse comportamento em um DLL, primeiro precisamos criar uma caverna de código dentro do DLL. Em seguida, precisamos modificar o código de operação no recurso de descrição do terreno para pular para a caverna de código no DLL.

3.4.4 Montagem em C++

Uma característica do C++ é a capacidade de inserir o código de montagem no arquivo de origem C++. Esta montagem não será modificada durante a etapa de compilação. Para isso, você pode usar a palavra-chave _asm. Por exemplo, o seguinte código pode ser usado para executar instruções pushad em arquivos de origem C++:

```
_asm{
Pushad
```

```
}
```

Você também pode misturar C++ e montagem em em uma função. Por exemplo, o código a seguir salva todos os registros, cria uma variável x, adiciona 1 a ela e, em seguida, restaura todos os registros:

```
_asm{
Pushad
}
int x=0; x=
x+1; _asm{

Bomba}
```

Por fim, as variáveis declaradas em C++ podem ser referenciadas nesses blocos de montagem. Usaremos esse comportamento mais tarde ao escrever programas de hacking.

3.4.5 Função de montagem

Para pular do código de Wesnos para nossa caverna de código, precisamos saber onde fica nossa caverna de código. A maneira mais fácil de fazer isso em C++ é declarar nossa caverna de código como uma função. Podemos então usar o operador & para recuperar seu endereço, o mesmo que outras variáveis. Seu pseudocódigo pode ser o seguinte:

```
void codecave(){
//Nossa Caverna de Código
}
...
Terrain_Description_Jump_Localização=&codecave;
```

Entretanto, durante a montagem, as funções geralmente são criadas usando uma estrutura de pilha. O quadro de pilha permite que o compilador desloque e calcule facilmente a localização de variáveis locais e parâmetros de função. À medida que exploramos pilhas, discutiremos esse comportamento mais profundamente em capítulos subsequentes. Para este capítulo, precisamos saber que a função de codecave acima será montada em:

```
Código Caverna:
Impulsionar ebp
mov ebp, esp
```

```
...
mov esp, ebp
pop ebp
direita
```

Quando saltamos para o jogo, esses comandos adicionais fazem com que nossas cavernas de código destruam o jogo. Esse dano pode fazer com que o jogo caia. Para evitar esse comportamento, usaremos a palavra-chave _declspec C++ para modificar a maneira como a função é montada. Quando esta palavra -chave é usada junto com a propriedade "naked", o compilador não adiciona uma estrutura de pilha.

3.4.6 Esqueleto da Caverna

Agora, podemos continuar criando nossa caverna de código. Primeiro, crie um DLL no Visual Studio, como fizemos no capítulo anterior. O projeto será chamado de Codecaavedll.

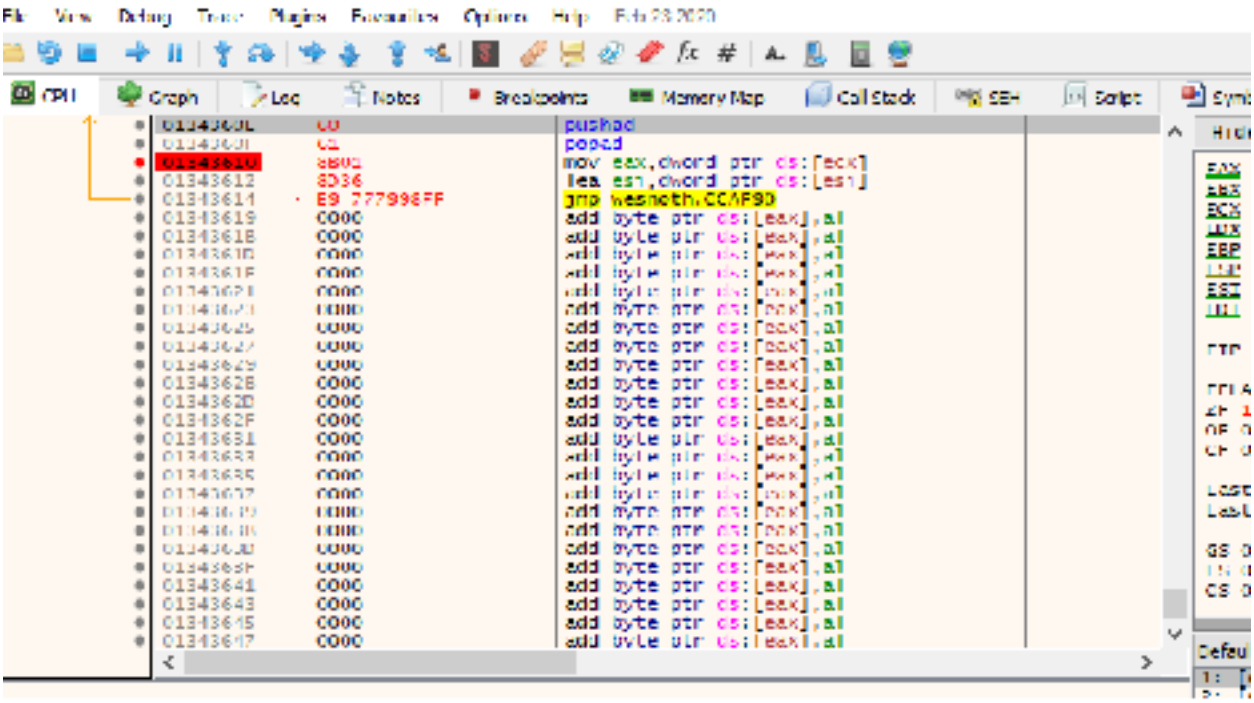
Depois de criar o DLL, podemos adicionar a função de caverna de código. Como discutimos acima, a função usará a palavra-chave __declspec para evitar que o compilador adicione uma estrutura de pilha. Sua definição é a seguinte:

```
__declspec (nua)void codecave(){}
```

Conforme discutido no Capítulo 2.6, o primeiro passo na criação de uma caverna de código é salvar e restaurar o registro e, em seguida, restaurar as instruções sobrepostas. Estabelecemos essas instruções no Capítulo 2.6.

```
Pushad
Rebote
mov eax, dword ptr ds: [ecx]
lea esi, dword ptr ds: [esi]
jmp 0xCCAF90
```

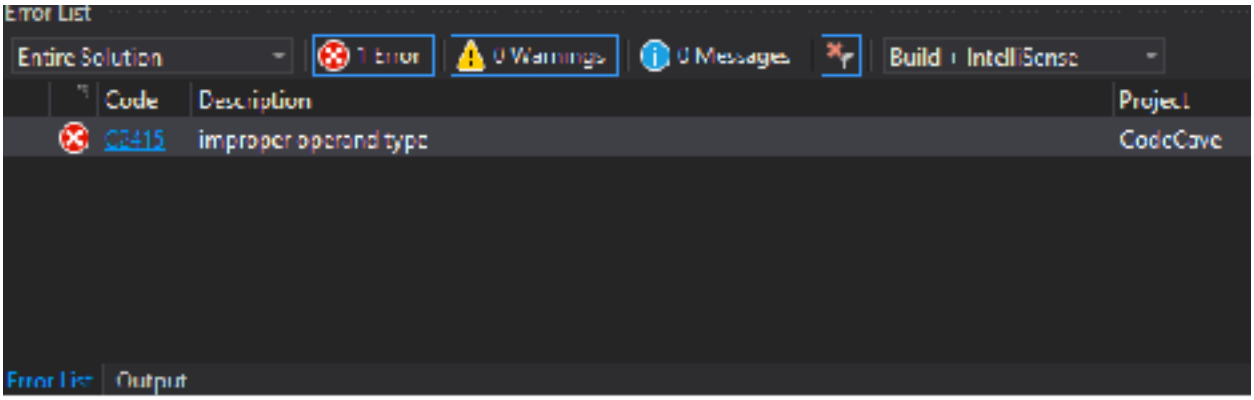
Quando esta caverna de código foi criada em x64dbg, ela parecia:



Em nosso DLL, criaremos dois blocos de instruções de montagem separados. O primeiro bloco salvará todos os registros. O segundo bloco restaurará o registro e executará as instruções originais que transcribimos. Entre esses dois blocos, colocaremos o código C++ para modificar o ouro do nosso jogador.

```
_asm{
Pushad
}

//Modificar o código do gold

_asm{
Rebote
mov eax, dword ptr ds: [ecx]lea
esi, dword ptr ds: [esi]jmp 0
xCCAF90
}
```

Se você tentar compilar este código, receberá um erro na instrução jmp:



Isso ocorre porque o compilador não consegue resolver instruções jmp quando um endereço estático é fornecido. Existem muitos tipos de instruções jmp que variam dependendo do comprimento do salto. Sem esse conhecimento, o compilador não saberia como codificar instruções. Existem várias maneiras de resolver essa ambiguidade , a mais simples das quais é criar uma variável. É isso que faremos neste capítulo.

Como nossa caverna de código não tem uma estrutura de pilha, não podemos declarar variáveis nela. Para contornar isso, declararemos todas as variáveis globalmente, bem abaixo da declaração include. Como precisamos salvar um valor de endereço estático, declararemos o endereço como DWORD:

```
#include<Windows.h>

DWORD Ret_Adress=0xCCAF90;

__declspec(naked)void codecave(){_
asm{
Pushad
}

//Modificar o código do gold

_asm{
Rebote
mov eax, dword ptr ds: [ecx]lea
esi, dword ptr ds: [esi]jmp Ret_
Adress
}}
```

3.4.7 Troca de Ouro

Depois de criar a função code cave, agora podemos usar o mesmo método discutido no Capítulo 3.3 para modificar o endereço dinâmico do gold usando alguns ponteiros.

Como discutimos na seção anterior, colocaremos esse código entre dois blocos de montagem para que nossa caverna de código possa salvar e restaurar corretamente todos os registros do jogo. O código usado para trocar nosso ouro será basicamente o mesmo do capítulo anterior. A única diferença é que primeiro precisamos declarar globalmente as variáveis fora da função code cave:

```
DWORD*player_base;
DWORD*gami_base;
DWORD* Ouro;
...
__declspec(naked)void codecave(){
_asm{
Pushad
}

player_base=(DWORD*)0x017EED18;
game_base=(DWORD*) (*player_base+0xA90); Good=
(DWORD*) (*tame_base+4);
*Ouro = 888; _
asm{

…
```

3.4.8 Redirecionamento

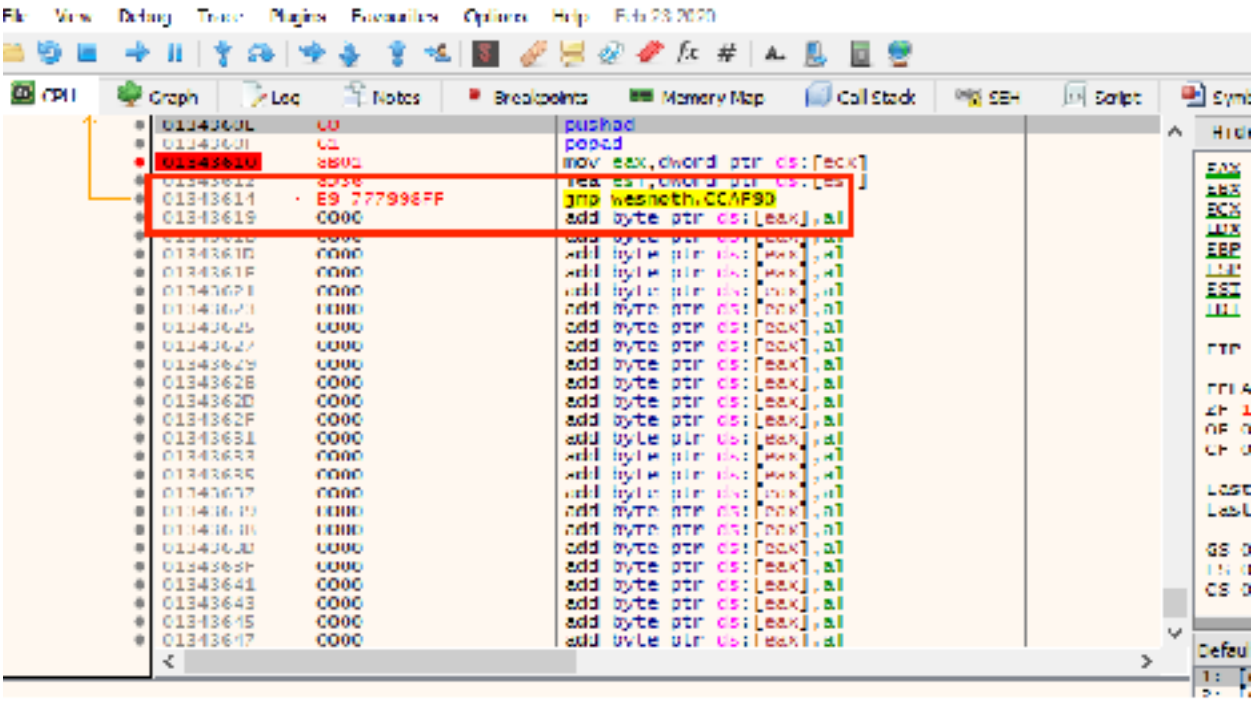
Em seguida, podemos redirecionar o código do jogo para chamar essa função. Para isso, usaremos o ponteiro novamente. Entretanto, desta vez, apontamos o ponteiro de declaração para o endereço do jogo responsável por exibir as características de descrição do Esta será a mesma posição de gancho que encontramos no Capítulo Cave de Código 0x00CCAF8A anterior.

Precisamos adotar uma abordagem um pouco diferente para modificar o código do jogo usando ponteiros. Como queremos modificar um único byte, declararemos o ponteiro como um char sem sinal (abreviação de um caractere). Ao contrário do DWORD, caracteres sem símbolos representam 1

Bytes de dados. Declarar nossos ponteiros dessa forma nos permitirá modificar um único byte com flexibilidade.

Antes de modificar o código do jogo, precisamos mudar seu tipo de proteção. O código é usado apenas para execução, portanto, por padrão, a Windows não permite que outros processos ou DLLs escrevam dados no endereço do código. Para alterar essa proteção, usaremos a API VirtualProtect e redistribuiremos os tipos de proteção.

Por fim, precisamos entender a estrutura do código de operação jmp. Sabemos que o valor do código de operação do jmp é 0xE9. Entretanto, há outros 4 bytes depois deste 0xE9. Esses bytes adicionais indicam para onde a CPU salta. Esses 4 bytes são mais do que apenas um novo endereço. Podemos ver um exemplo na caverna de código criada anteriormente usando x64dbg:



Existem vários recursos on-line que descrevem como esse código de operação é construído. A fórmula básica é:

```
NeW_Localização-Originação+5
```

Vamos verificar esta fórmula usando a caverna de código acima:

```
0xCCAF90-0x1343614+5=FF 98 79 77
```

Isso parece incorreto no início. Entretanto, em todas as CPUs compatíveis com Windows, os bytes são armazenados em ordem "oposta". Este é um conceito chamado Endianese, que será apresentado em detalhes em capítulos subsequentes. Se invertermos a ordem de bytes do valor encontrado acima, descobriremos que ele corresponde ao código de operação em x64dbg:

```
77 79 98 FF
```

Agora que verificamos que essa fórmula é válida, podemos implementá-la em nosso próprio código, jmp em nossa caverna de código.

Função de redireção 3.4.9

Quando um DLL é injetado pela primeira vez, processamos a redirecionação na função DllMain. Primeiro, precisamos declarar um ponteiro apontando para a posição do gancho. Além disso, a API VirtualProtect requer um parâmetro para salvar o tipo de proteção anterior. Também declararemos:

```
DWORD old_protect;
unsigned char*Hook_Localização=(unsigned char*)0x00CCAF8A;
```

Em seguida, mudaremos o tipo de proteção para a posição do gancho. A API VirtualProtection tem parâmetros semelhantes aos APIs ReadProcession e WriteProcession. Como fizemos no capítulo anterior, precisamos reescrever 6 bytes.

```
Se(fdwReason==DL_Process_Attach){
Projeto Virtualítico ((void*)Hook_Locasionismo, 6. Página _
Locasionismo, &ol_Projeto);
//
Redirecionamen
to}

Retornar a verdade;
```

Agora que a localização está escrita, podemos começar o processo de redistribuição de bytes para pular para nossa caverna de código. Primeiro, definimos o primeiro byte para 0xE9:

```
*Hook_Localização=0xE9;
```

Em seguida, usaremos a fórmula testada acima para escrever o código de operação adicional necessário para o jmp. Esses códigos de operação começarão 1 byte após a posição do gancho:

```
*(Hook_Locação+1)=&codecave-(Hook_Locação+5);
```

Entretanto, esse código não funcionará conforme o esperado. Isso gravará apenas 1 byte, não 4 byte. Isso ocorre porque o hook_localização é definido como um ponteiro apontando para um caractere sem sinal com 1 byte de comprimento. Para gravar os 4 bytes que precisamos, converteremos hook_localização em um ponteiro apontando para DWORD. Também converteremos outras variáveis em DWORD:

```
*(DWORD*)(hook_localização+1)=(DWORD)&codecavidade-((DWORD)hook_localização+5);
```

Por fim, como fizemos no capítulo anterior, precisamos tornar o sexto byte nop. Isso pode ser feito da mesma maneira que usamos para definir o primeiro byte como jmp. Adicionamos 5 (em vez de 6) porque o valor em C++ é indexado a partir de 0:

```
*(hook_localização+5)=0x90;
```

Uma vez concluído, podemos construir e injetar DLL como fizemos no capítulo anterior. No jogo, selecione a descrição do terreno em qualquer azulejo. Antes de exibir a descrição, seu ouro deve ser definido para 888.

Os códigos completos são encontrados no Apêndice A para comparação.

3.5 Texto de impressão

3.5.1 Objetivo

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

3.5.2 Identificação

Neste capítulo, imprimiremos nosso próprio texto usando o Wesnoth. Para concluir essa tarefa, primeiro encontraremos um pedaço de código responsável pela impressão do texto. Em seguida , usaremos uma caverna de código para modificar a memória do jogo para exibir nosso texto.

3.5.3 Compreensão

Existem muitas maneiras de imprimir nosso próprio texto no jogo:

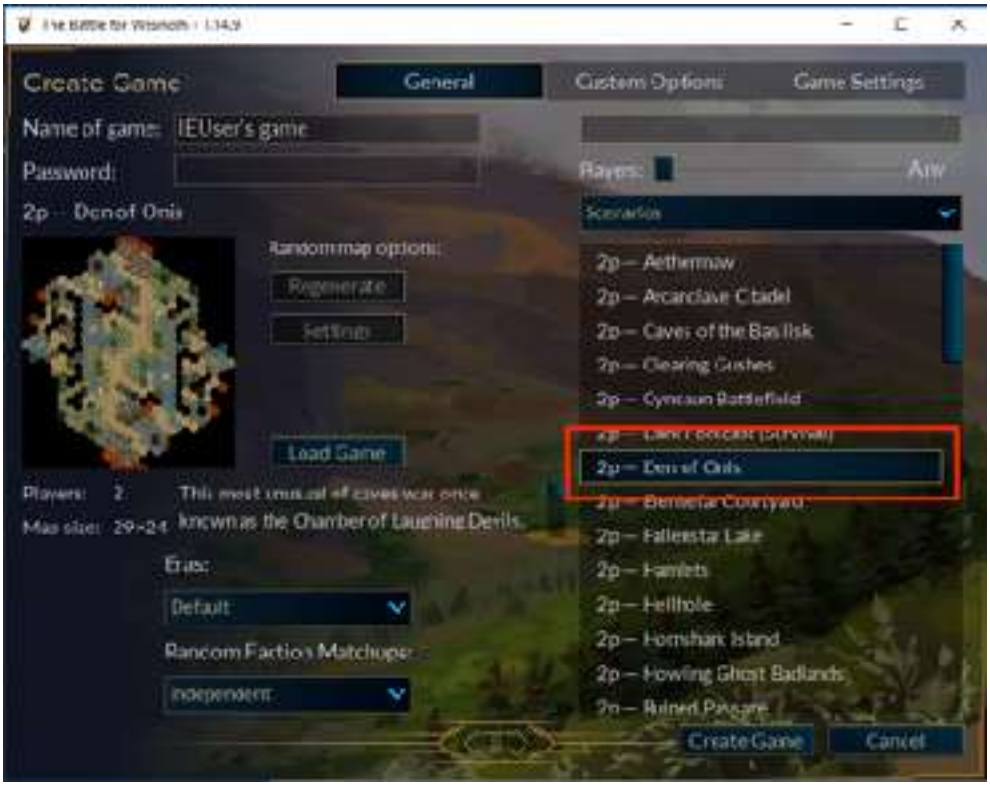
1. Use cobertura externa.
2. Crie uma caverna de código dentro do loop principal de exibição do jogo e chame a função responsável por exibir o texto.
3. Crie uma caverna de código dentro de uma função responsável por exibir texto e modifique o texto a ser exibido.

Jogos diferentes são mais adequados para métodos diferentes. Neste capítulo, usaremos a terceira abordagem, que é a maneira mais fácil de fazer em Wesnoth. Exploraremos mais profundamente outras abordagens em capítulos subsequentes.

3.5.4 Texto de localização

Nossa primeira tarefa foi encontrar o código do jogo responsável por exibir o texto. Primeiro, precisamos encontrar uma sequência de letras que aparece no jogo. Para o Wesnoth, podemos usar o texto de descrição do terreno exibido ao clicar no bloco. Qualquer descrição é aceitável, mas neste capítulo usaremos a descrição da Ford Tiles. Para outros jogos, mensagens de bate-papo são um bom ponto de partida.

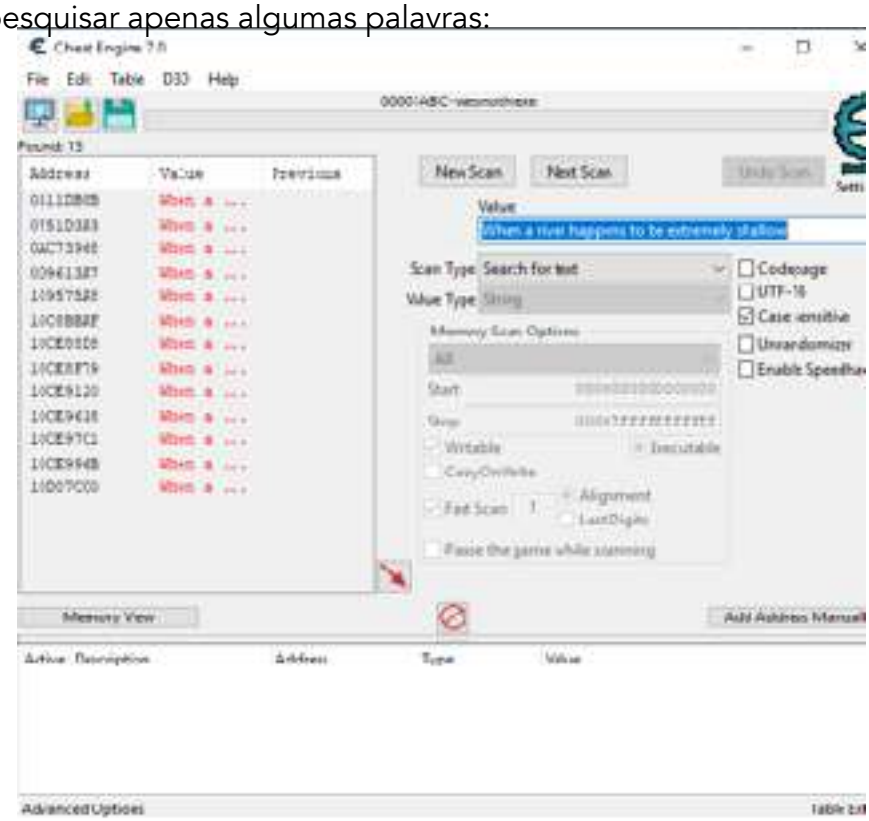
Primeiro, escolha um mapa com azulejos Ford nele. O ninho de Onis é um exemplo:



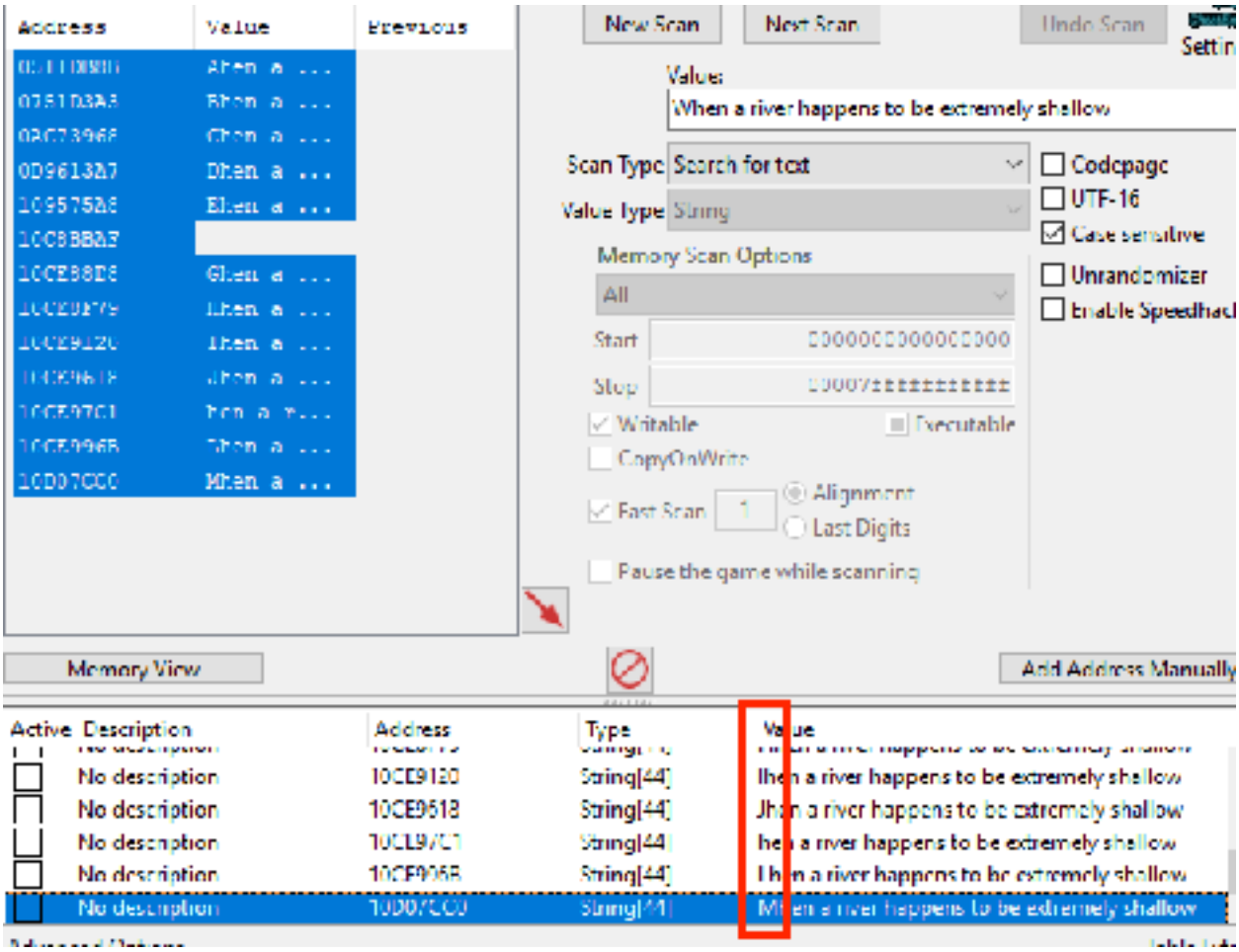
Quando o mapa estiver carregado, selecione a Colagem Ford e, em seguida, selecione a entrada de descrição do terreno no menu contexto:

Isso exibirá uma descrição do adesivo, que contém uma longa sequência de texto:

Podemos usar o mecanismo de fraude para pesquisar onde esse texto está armazenado no jogo. Certifique-se de fechar a caixa de descrição do terreno antes de pesquisar para reduzir o número de resultados. Devido à singularidade do texto, precisamos pesquisar apenas algumas palavras:

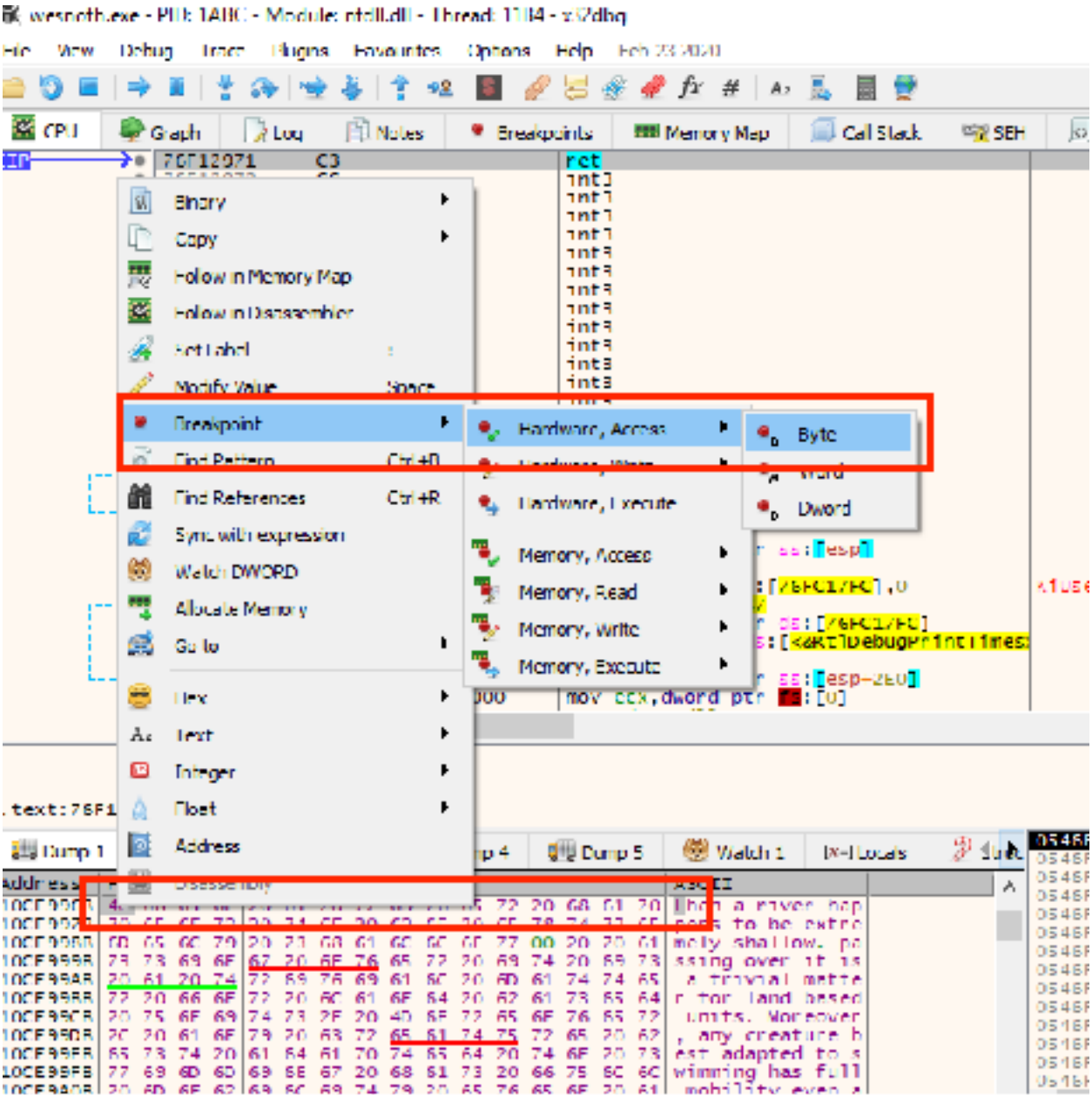


Para reduzir o endereço da string que nos interessa, mude a primeira letra de cada string. Após alterar esses valores, retorne a Wesnoth e verifique a descrição do terreno novamente. A versão da string exibida no jogo corresponderá ao endereço correto. Neste exemplo, uma string que começa com Lhen é exibida, de modo que nosso endereço seja 0x10CE996B.

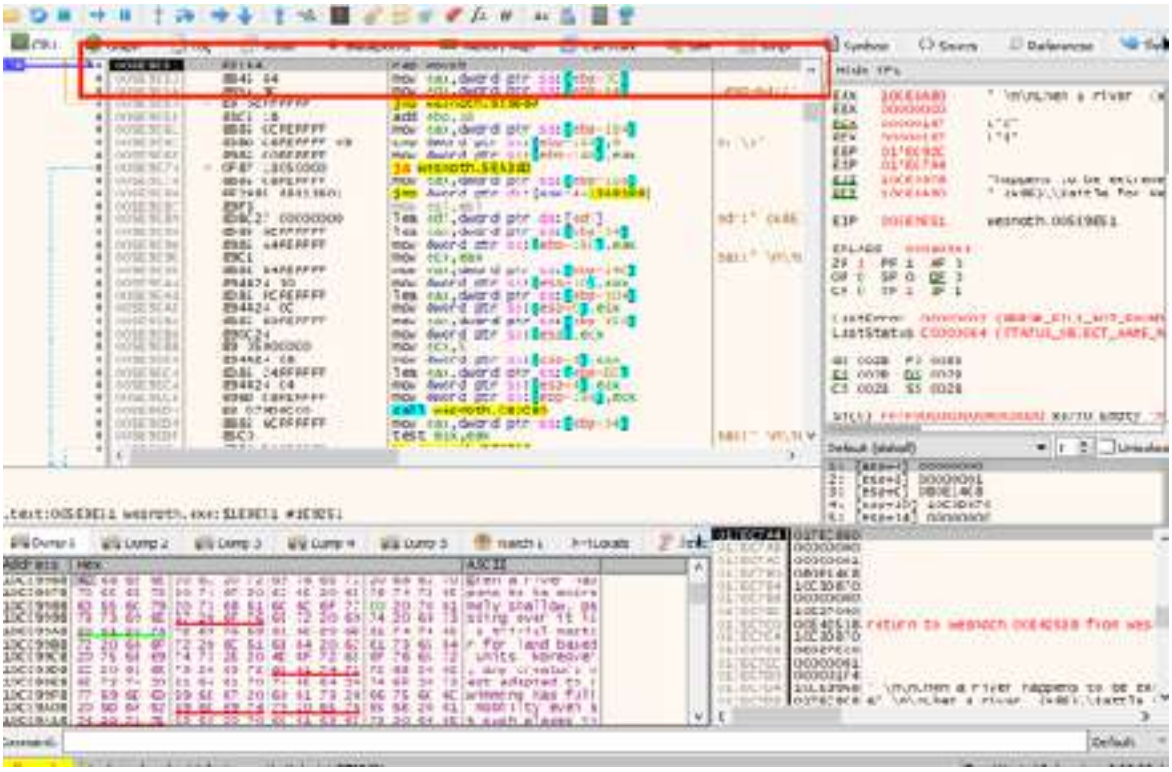


3.5.5 Posicionamento PrintText

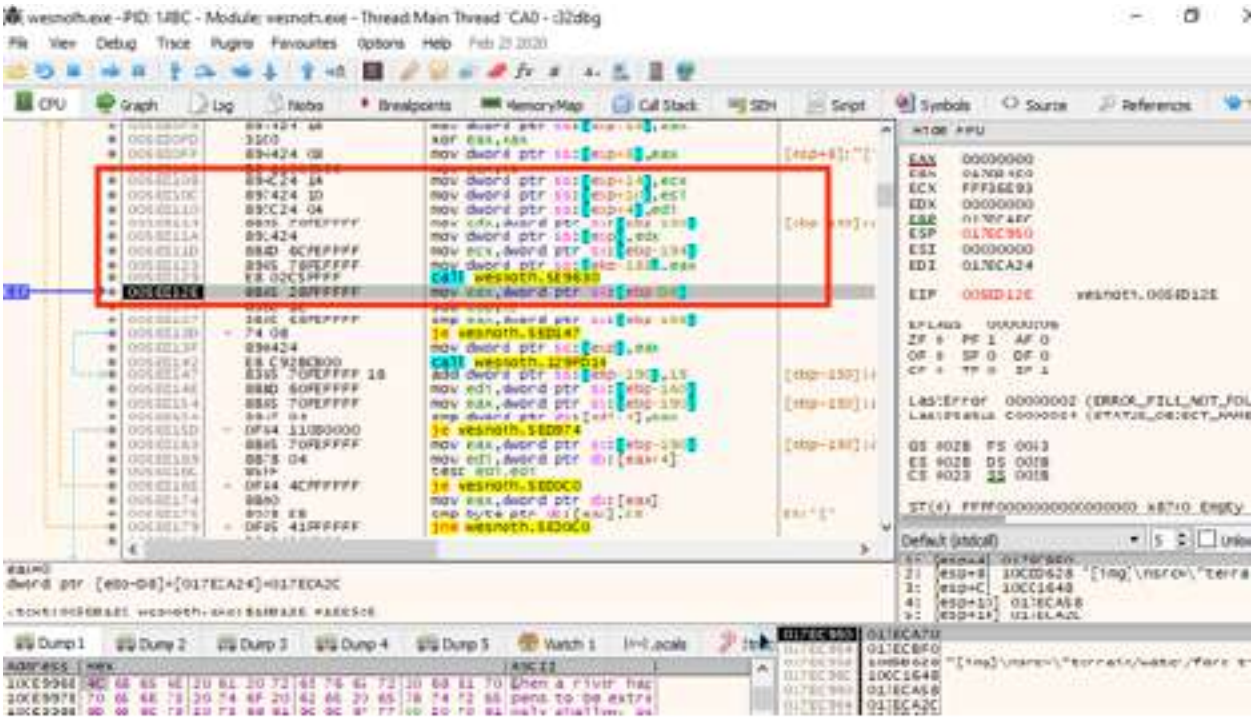
Agora podemos usar o endereço encontrado para localizar a função responsável pela impressão do texto. Sabemos que a função print text deve acessar o texto de alguma forma para imprimi-lo. Para determinar a localização desta função, podemos definir um ponto de interrupção em um byte do texto.



Após definir o ponto de interrupção, retorne a Wesnoth e chame a operação de descrição do terreno novamente. Seu ponto de interrupção aparecerá imediatamente:



Verificando esse código, parece que estamos em um loop responsável por mover cada byte do texto para o buffer. Como fizemos em capítulos anteriores, queremos navegar para o código que chama esse código de baixo nível usando o Excute until Retour e o stepping out.



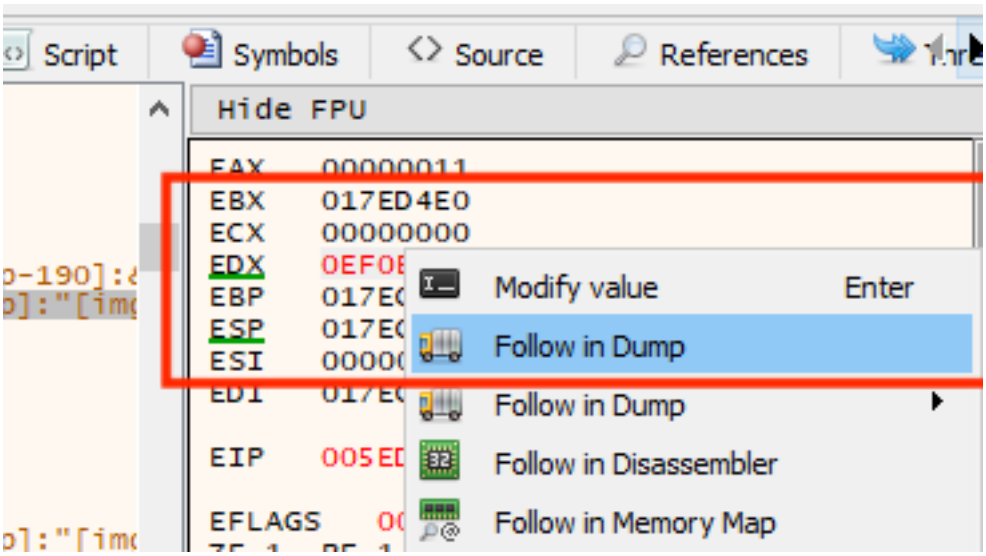
Esta chamada pode parecer responsável por preencher a caixa de descrição do terreno com texto. Se continuarmos executando, notaremos que esse código foi chamado várias vezes para cada parte da caixa de descrição do terreno. Para determinar os parâmetros passados para esta chamada, podemos definir um ponto de interrupção em 0x005ED114 e chamar a ação de descrição do terreno novamente:

Parte do texto de exibição destacado é carregado no registro edx. O código em 0x005ED11A então move o valor em edx para a posição apontada pelo esp. Isso é o mesmo que empurrar o valor do edx para o topo da pilha. Embora ainda não tenhamos discutido pilhas, para o propósito deste capítulo, precisamos saber que as funções normalmente recuperam valores da pilha para execução.

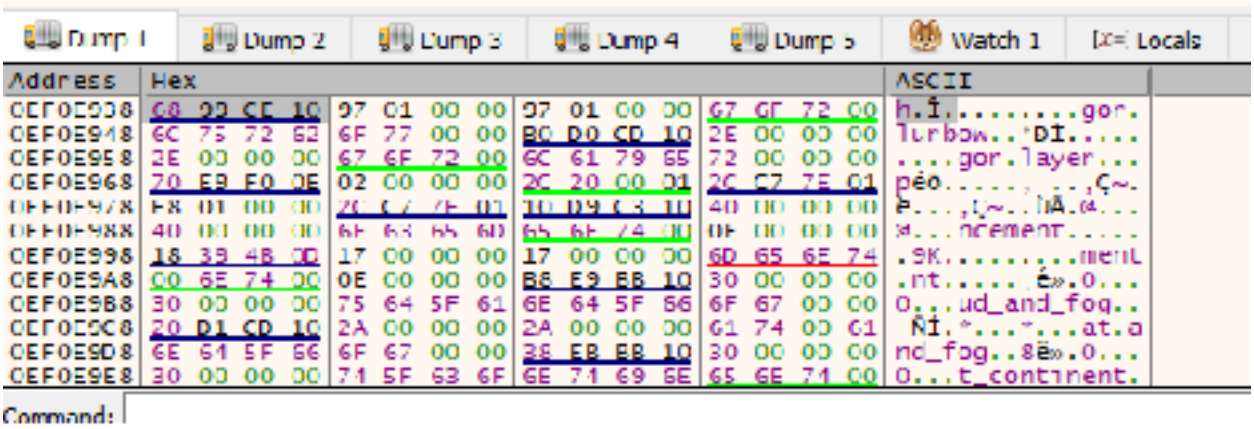
3.5.6 Memória e ordem de bytes

Você pode ter notado que o endereço no edx não corresponde ao endereço que encontramos no cheating Engine. Como o espaço de texto é alocado dinamicamente, precisamos entender como recuperar valores de texto do edx para criar nossas cavernas de código.

Chame a operação de descrição do terreno novamente para forçar nosso ponto de interrupção a aparecer. Quando concluído, clique com o botão direito no valor edx e selecione Follow in Dump:



Isso alterará o endereço atual exibido no dump para o valor edx. Como discutimos em capítulos anteriores, a seção de dump mostra a memória executiva atual do processo. É importante lembrar que tanto a seção dump quanto o mecanismo de fraude exibem os mesmos dados que pesquisam.



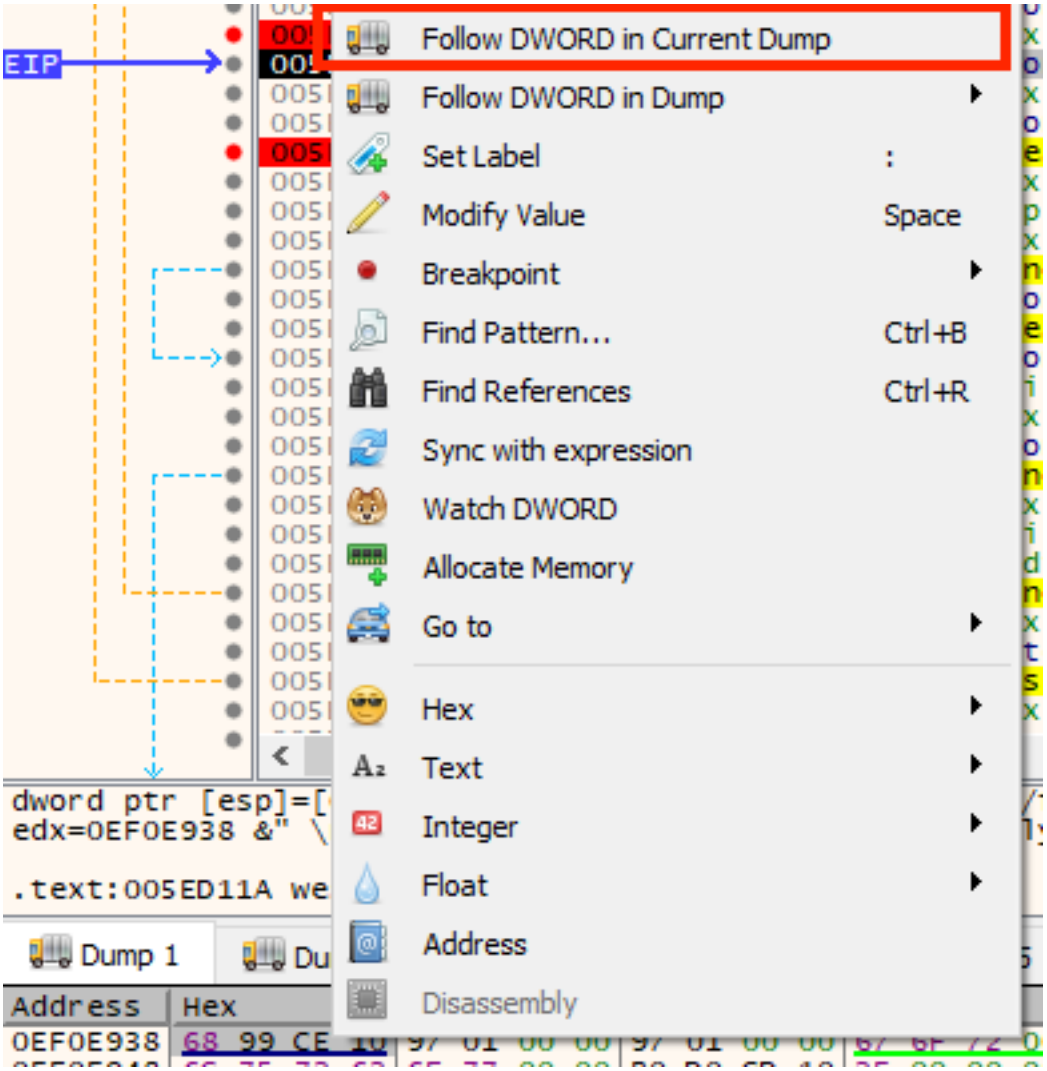
Este valor armazenado em edx obviamente não é uma string de texto. No entanto, se verificarmos os valores dos bytes, descobriremos que eles têm muitas semelhanças com os endereços que encontramos no mecanismo de fraude. No capítulo anterior, discutimos brevemente um conceito chamado ordem de byte. A maioria das CPUs baseadas em Windows são pequenas. Por definição , isso significa que os bytes menos válidos são armazenados no menor endereço.

Na verdade, isso significa que quando o endereço 0x12345678 for armazenado na memória, ele será armazenado como 0x78 56 34 12. Neste caso, 0x78 representa o menor byte válido ou o valor mínimo. Uma boa comparação é imaginar o número 123. Expresso como

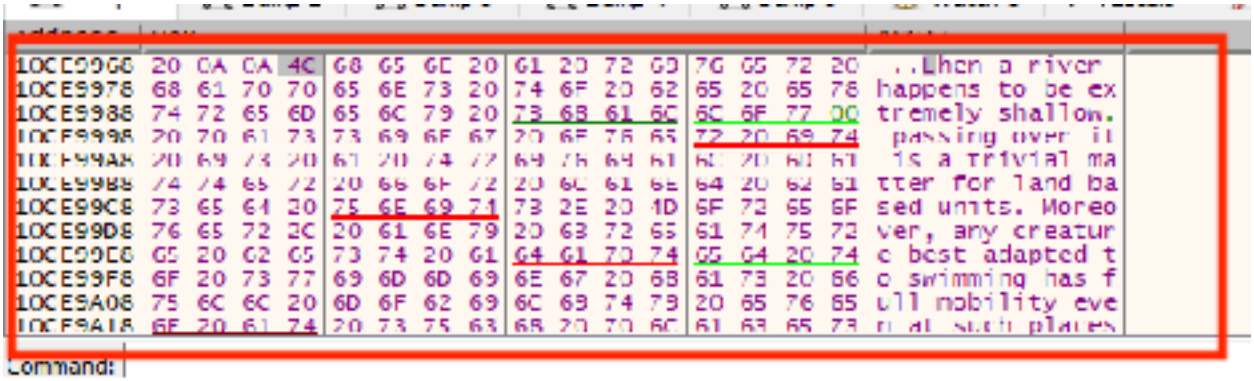
Em um formato mais longo, esse valor pode ser entendido como 1*100+2*10+3*1. O valor mínimo nesta forma é o número 3.

A segunda parte desta definição pode ser entendida verificando o dumping. No dump , o endereço de memória aumenta de um valor menor para um valor maior. Portanto, o byte mínimo válido é armazenado "primeiro" na memória. A combinação desses fatores faz com que os endereços armazenados na memória pareçam "invertidos".

Agora que entendemos a ordem de bytes, podemos concluir que o valor armazenado em edx é um endereço. Podemos navegar rapidamente para este endereço no Dump selecionando todos os bytes e selecionando Follow in Dump novamente:



Após selecionar esta opção, chegamos ao local da string na memória:



Para referir esse valor na assembleia, podemos usar a palavra-chave ptr ds:

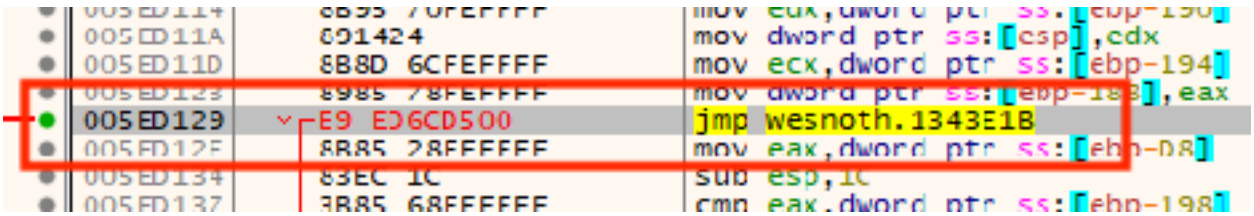
```
mov eax, dword ptr ds: [edx]
```

Isso carregará o valor do endereço armazenado em edx no eax. Neste caso, ele carregará o valor 0x10CE9968 no eax. Podemos então usar a palavra-chave ptr ds novamente para acessar cada byte do texto.

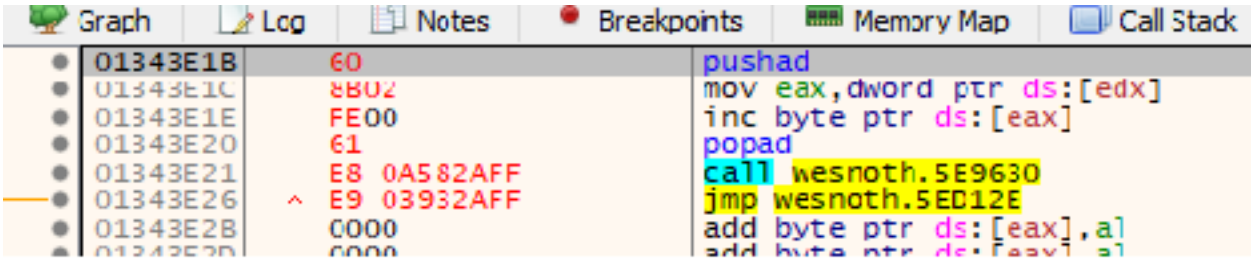
3.5.7 Alteração de texto

Uma vez concluído, podemos começar a criar nossos hackers. Para verificar que temos a abordagem correta, criaremos uma caverna de código que alterará o texto exibido toda vez que uma operação de descrição do terreno for chamada. Para fazer isso facilmente, adicionaremos o valor do primeiro byte a cada vez que executarmos a caverna de código. Isso mudará o valor dos personagens e nos permitirá confirmar que nossos hackers estão trabalhando.

Como sabemos que a chamada em 0x005ED129 é responsável por imprimir texto e também tem 5 bytes de comprimento, usaremos isso como ponto de redirecionamento. Conforme discutido no capítulo anterior sobre cavernas de código, qualquer local próximo ao fim da memória do programa se aplicará à nossa localização de cavernas. Neste caso, nós o criaremos em 0x01343E 1B. Como de costume, substituiremos a posição do gancho saltando para a caverna do código:



Primeiro, salvaremos os registros na caverna de código. Em seguida, usaremos a palavra-chave ptr ds para carregar o valor do texto de edx para eax. Depois, usaremos o operador inc para aumentar o valor do primeiro byte da string. Por exemplo, se o primeiro byte for atualmente a (valor ASCII 65), ele aumentará para B (valor ASCII 66). Por fim, restauramos o registro, recriamos a chamada e retornamos ao código original.



Após concluído, retorne a Wesnoth e chame a operação de descrição do terreno várias vezes. Você notará que um caractere atrás da imagem muda a cada vez, o que indica que modificamos com sucesso o texto exibido.

Parte 4
Hacker RTS

4.1 STATACK

4.1.1 Objetivo

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

4.1.2 Identificação

Neste capítulo, criaremos um hacker estatístico, comumente conhecido como stathack. Esse tipo de hacker nos mostra informações sobre outros jogadores, como seu ouro ou número de unidades.

Neste capítulo, nosso Stattack exibirá o ouro do segundo jogador.

4.1.3 Compreensão

Para criar o stathack, precisamos concluir duas etapas:

1. Encontre o ouro do segundo jogador.
2. Imprima esse valor na tela.

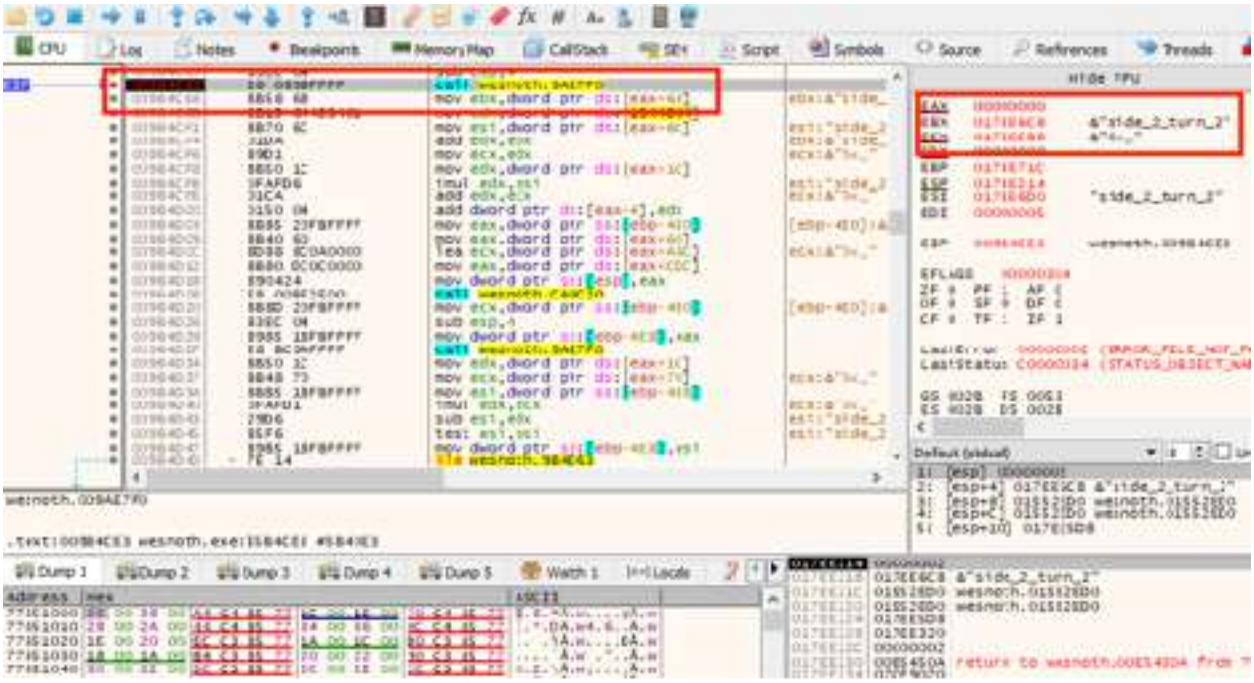
No capítulo anterior, apresentamos as técnicas para concluir essas duas etapas.

4.1.4 Ouro do Segundo Jogador

Voltando ao Capítulo 1.2, exploramos como os jogos frequentemente atribuem dados e classes semelhantes em matrizes. O jogo então pode iterar esses array para localizar e atualizar os dados. Embora não saibam se Wesnoth trabalha dessa maneira, podemos usá-lo como um modelo para tentar identificar o valor ouro do segundo jogador.

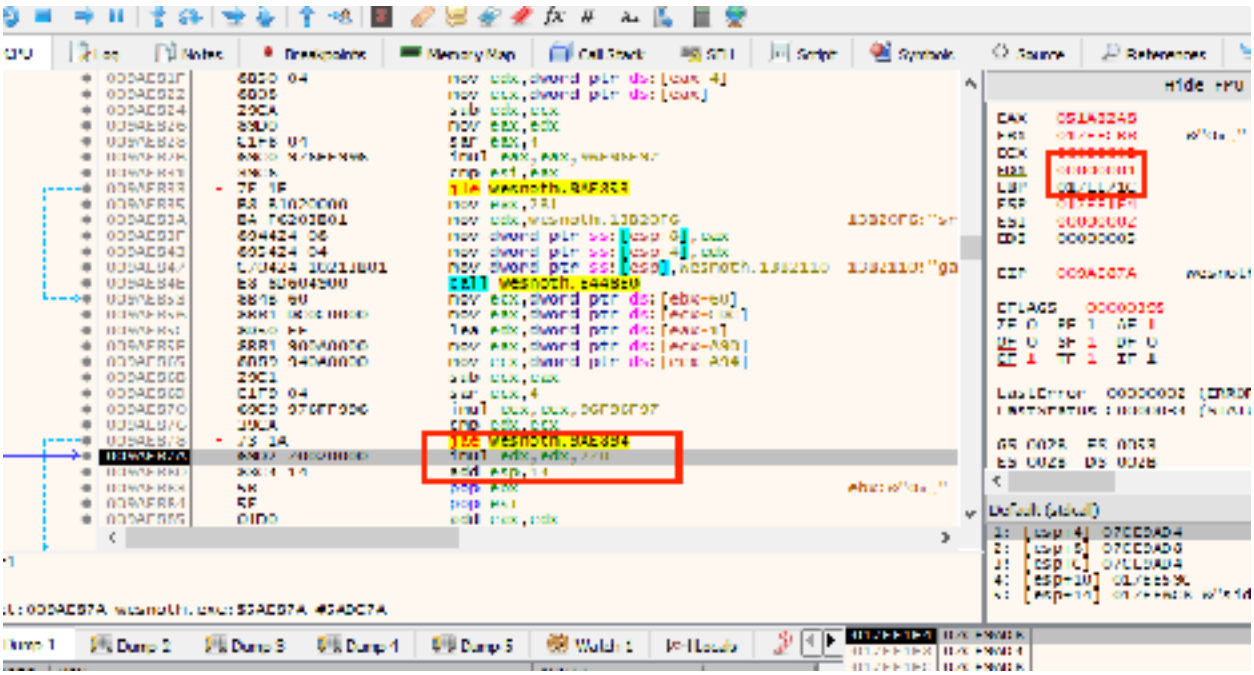
Sabemos no Capítulo 2.8 que o jogo atribui dinamicamente as classes de jogadores com base em um ponteiro básico. Também determinamos os ponteiros básicos para o jogo e o primeiro jogador. Ao examinar cuidadosamente o código que encontramos neste capítulo, podemos determinar se o jogo usa um array para sua classe de jogadores e, se sim, localize o ponteiro básico do segundo jogador.

Primeiro, crie um jogo local com dois jogadores locais. Certifique-se de que ambos os jogadores ganhem renda por rodada. Inicie o jogo com x64dbg anexado. Jogue uma rodada para cada jogador para garantir que qualquer código de inicialização da primeira rodada tenha sido executado. Em seguida, defina um ponto de interrupção no x64dbg no 0x9B4CE3, o mesmo chamado que determinamos anteriormente. No Wesnoth, após terminar a segunda rodada do primeiro jogador, o ponto de interrupção deve aparecer:

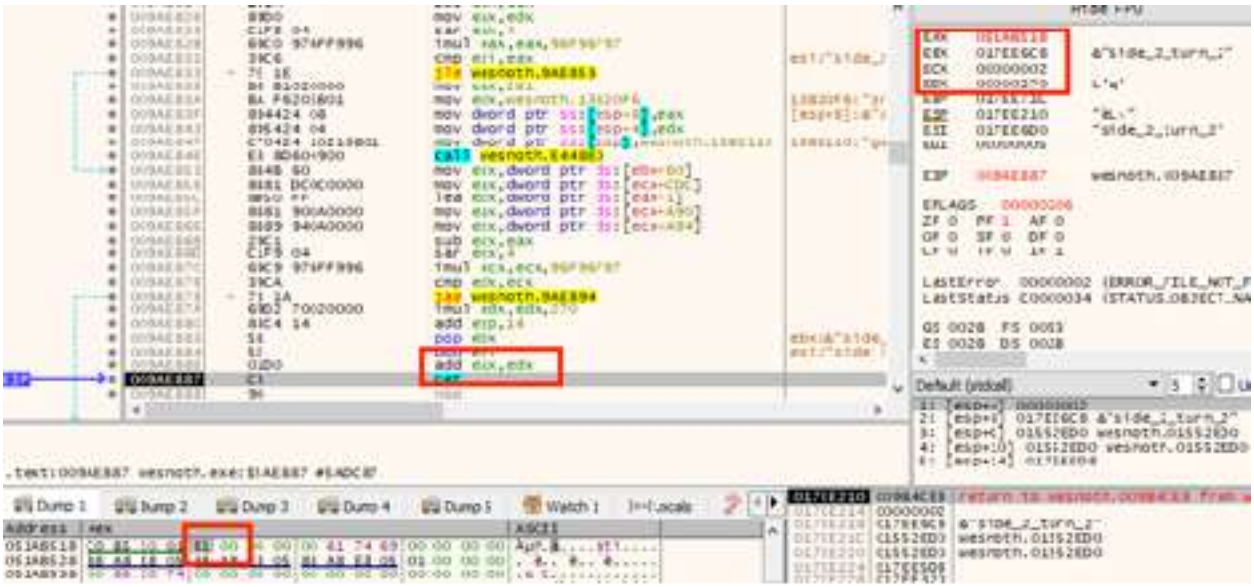


Os valores em ebx indicam que cada jogador chamará essa função a cada turno. Além disso, sabemos que o valor no ecx é o ponteiro básico do jogo. Com base nesses dois fatos, podemos presumir que o jogo tem um array semelhante ao jogador.

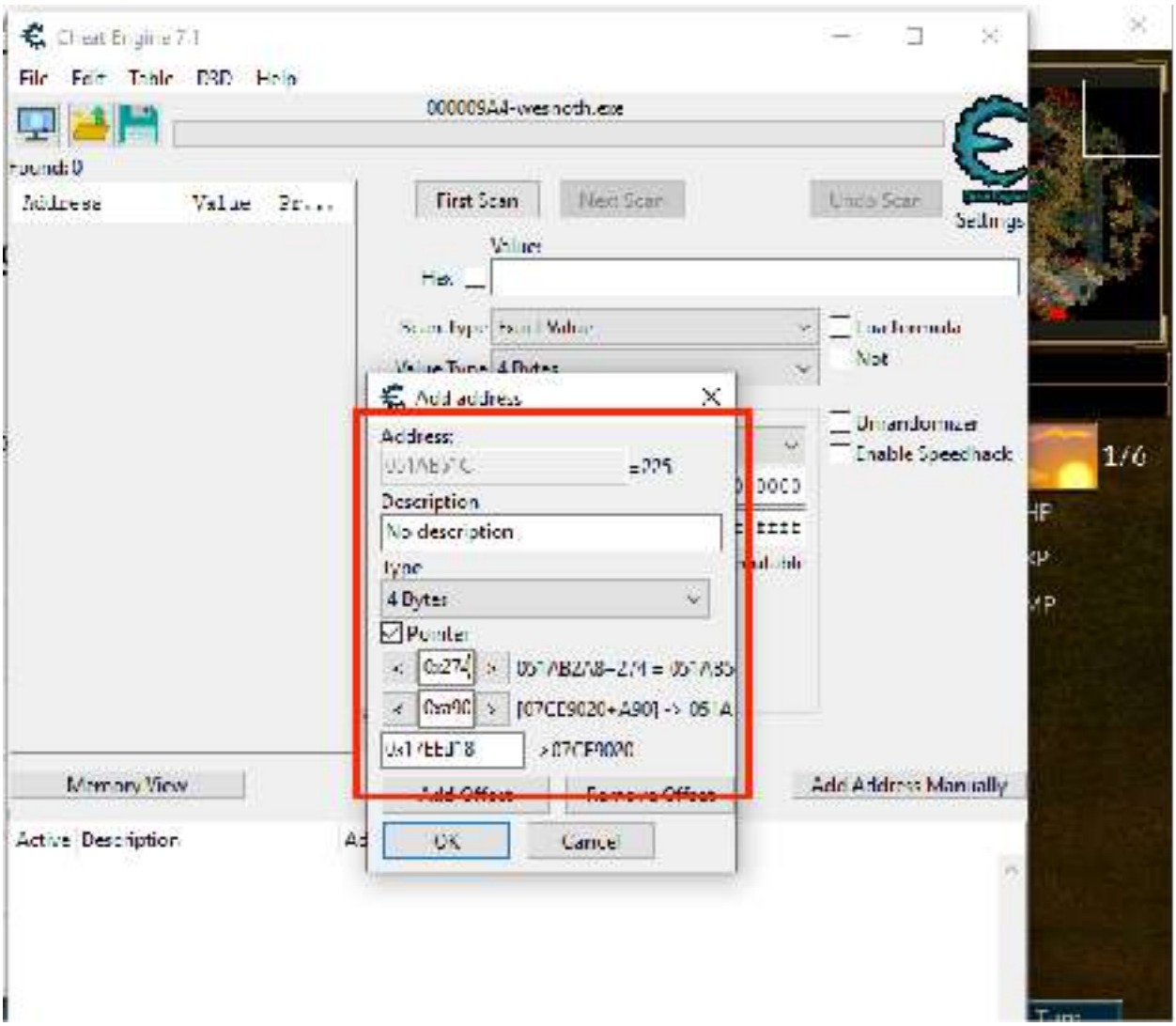
Nosso próximo passo é determinar o tamanho de cada jogador no array. O jogo precisa saber esse tamanho para avançar até o próximo jogador no array. Insira a chamada em 0x9B4CE3 e conclua cada linha de código passo a passo. Como sabemos no capítulo anterior, este código devolverá o endereço dourado do jogador no eax. Para a maior parte desta função, os endereços e valores usados são os mesmos que observamos ao procurar o endereço dourado do primeiro jogador. Entretanto, há uma instrução imul (multiplicação sinalizada) próxima à parte inferior da função:



À medida que completamos essa função passo a passo como o primeiro jogador, edx é definido para o valor 0. Entretanto, à medida que passamos pelo segundo jogador, o edx é definido para o valor 1. Como pode ser visto neste código, o jogo usa edx para deslocar o jogador atual e, em seguida, multiplicar o edx por um valor 0x270 para identificar o jogador atual. Se seguirmos algumas linhas de código abaixo, veremos que esse valor foi adicionado ao valor eax para obter o endereço dourado do jogador atual:



A partir deste código, podemos determinar como compensar o valor do ouro do nosso segundo jogador. Como descobrimos anteriormente, usaremos [[0x017EED18]+0xA90] para deslocar o ponteiro básico do jogo. Se adicionarmos 4, receberemos o endereço dourado do primeiro jogador. Para obter o endereço dourado do segundo jogador, podemos adicionar 0x270+4 ou 0x274. Podemos usar o mecanismo de fraude para verificar esse cálculo:



Nos capítulos anteriores, escrevemos código para compensar o endereço dourado do primeiro jogador. Podemos modificar este código usando o novo valor 0x274 para recuperar o endereço dourado do segundo jogador, conforme mostrado abaixo:

```
player_base=(DWORD*)0x017EED18;
game_base=(DWORD*)(*player_base+0xA90);
gold=(DWORD*)(*gami_base+0x274);
```

4.1.5 Valor de impressão

No capítulo anterior, apresentamos um método de impressão de texto. Determinamos que, ao criar uma caverna de código, podemos acessar o texto do método de descrição do terreno referindo-se aos valores apontados por edx. Depois de acessá-lo, podemos armazenar bytes neste local, exibidos pelo jogo.

Usando o método do Capítulo 3.4, podemos implementar isso em DLL. Como discutimos métodos para redirecionar código, agora examinaremos apenas funções de caverna de código. Começaremos com o esqueleto:

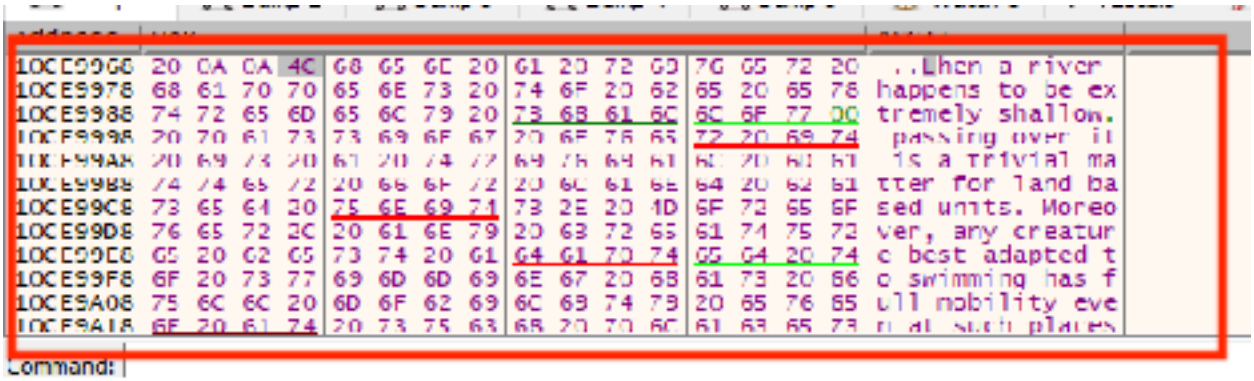
```
DWORD or_Call_adress=0x5E9630; DWORD
Ret_Adress=0x5ED12E;

__declspec(naked)void codecave(){_
asm{
Pushad
}

//Novo código

_asm{
Rebote
Chamar ORI_Call_Adress jmp
Ret_Adress
}}
```

Já vimos esse código antes. A principal diferença é que as instruções que substituímos para impressão de texto são uma chamada.

Este código cave é chamado toda vez que um método de descrição do terreno é chamado. Entre eles, queremos pesquisar o valor do ouro do segundo jogador. Podemos fazer isso usando o código discutido na seção anterior:

```
_asm{
Pushad
}

player_base=(DWORD*)0x017EED18;
game_base=(DWORD*) (*player_base+0xA90);
gold=(DWORD*) (*gami_base+0x274);
```

Agora armazenamos o valor de ouro do segundo jogador na variável de ouro. Para exibir esse valor no jogo, precisamos convertê-lo em uma string. Para entender as tarefas que estamos tentando realizar, aqui está um dump de memória que contém a string de texto que encontramos no capítulo anterior:



Observando isso, podemos ver que mesmo que o jogo exiba Lhen, o valor armazenado na memória é 0x4C 68656e. Isso ocorre porque o jogo usa codificação ASCII para codificar valores de caracteres em determinados números. O jogo então sabe como decodificar esses valores e exibir os personagens correspondentes no jogo.

Então, se o valor áureo que queremos exibir é 225, não podemos simplesmente gravar 225 no jogo e esperar que o jogo seja bem-sucedido. Em vez disso, precisamos convertê-lo para 2 2 5, ou 0x32 32 35. Existem várias maneiras de fazer isso, a mais simples é usar a API sprintf_s:

```
#Incluindo <stdio.h>

char Gold_Bytte_Array[4]={0};

...
gold=(DWORD*)(*gami_base+0x274); sprintf_s(
gold_byte_array,4, "%d",*gol);
```

Isso converte o valor apontado pela variável gold em sua representação de string e armazena o valor em gold_byte_array.

Por fim, moveremos esse valor convertido para a memória a ser exibida. Como o sprintf_s alterou vários registros, primeiro os restauramos e os salvamos novamente para garantir que o jogo não travesse:

```
_asm{
Rebote
Pushad
mov eax, dword ptr ds: [edx]
```

No capítulo anterior, simplesmente aumentamos o primeiro caractere apontado por eax. Para exibir o valor do gold, movemos o valor armazenado em gold_byte_array:

```
mov bl, Gold_Byt_Array[0]
mov byte ptr ds: [eax], bl
```

Primeiro, movemos o primeiro byte do gold_byte_array para o bl. Para entender por que usamos bl, precisamos entender os diferentes tamanhos de dados usados pela CPU. Um bit é a menor unidade de dados, representando 0 ou 1. Um byte é 8 bits. Uma palavra é 16 dígitos. A palavra dupla (ou DWORD) é 32 bits. Ao visualizar o dump de memória em x64dbg, vemos o valor do byte. Os quatro desses valores de byte são combinados para formar um DWORD.

Atualmente, todos os registros que vemos, como o ebx, são DWORD. As primeiras CPUs Intel, como o 8088, usavam registradores de 16 bits ou palavras, como o bx. Para acessar cada byte no registro bx, você pode usar h e l, como bh e bl. Nas CPUs modernas, essas mesmas regras se aplicam mesmo se usarmos o formato estendido (ou DWORD) desses registros. Como estamos movendo bytes individuais, precisamos movê-los para um valor que possa acomodar um byte. Em seguida, movemos o valor para o local apontado pelo eax, que também é um byte de comprimento.

Este código moverá nosso primeiro caractere para o texto. Podemos repetir isso várias vezes para mover personagens adicionais. Para este capítulo, exibiremos apenas 3 caracteres de dados:

```
mov bl, Gold_Bytte_Array[1]mov
Bytte_Array ds: [eax+1], bl mov
bl, Gold_Bytte_Array[2]mov
Bytte_Array: [eax+2], bl
```

Por fim, como no Capítulo 3.4, redirecionaremos a função de texto impresso do jogo para a caverna de código no DllMain:

```
DWORD old_protect;
unsigned char*Hook_Localização=(unsigned char*)0x5ED129;
```

```
Se(fdwReason==DL_Process_Attach){
Projeto Virtuo* (void*)Hook_Locasione, 5. página _Express_Readwrite,
&old_protect);
*Hook_Localização=0xE9;
*(DWORD*)(hook_localização+1)=(DWORD)&codecavidade-((DWORD)hook_localização
+5);
}
```

Construa e injete-o da mesma maneira que fizemos nos capítulos anteriores. Por fim, entre no jogo e abra a descrição do terreno nos azulejos. Devido à forma como ganhamos a função, devemos ver que o valor dourado do segundo jogador foi impresso várias vezes:

O código completo para este capítulo está disponível no Apêndice A.

4.2 Hacker de Mapas

4.2.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

4.2.2 Identificação

Nosso objetivo neste capítulo é criar um hacker de mapas, um hacker que exibe o mapa inteiro ao jogador e remove elementos como a névoa de guerra.

4.2.3 Compreensão

Em jogos de estratégia como o Wesnoth, os azulejos no mapa podem ser visíveis ou escondidos na névoa de guerra:

Sabemos se o jogo deve armazenar azulejos em algum lugar da memória para ver se eles são visíveis. Esses locais provavelmente estão em um grande bloco de memória. Uma maneira que o jogo pode escolher representar um mapa é usar um array. Neste array, cada elemento representa o estado de visibilidade de um bloco de mapa:

```
map [MAP_tamanho]={0,0,1,0,0,1,1,1,1,0, …}
```

O jogo então pode iterar cada bloco no conjunto para determinar se uma neblina deve ser desenhada no bloco.

Também sabemos que sempre que um jogador move uma unidade, o jogo deve calcular o valor de cada carta. Se um jogador mover uma unidade dentro do alcance de uma carta, o jogo precisa definir a visibilidade da carta como verdadeira. Se um jogador remover uma unidade do alcance da carta, o jogo precisa definir a visibilidade da carta para falsa. Para facilitar esse cálculo, os jogos geralmente definem todos os blocos como invisíveis primeiro:

```
for (adesivos em adesivos
){adesivos [adesivos]=0

}
```

O jogo então permite que todos os azulejos ao redor fiquem visíveis através de cada unidade controlada pelo jogador.

Embora cada jogo tenha sua própria maneira de lidar com dados do mapa, todos eles devem seguir um conjunto de etapas semelhantes para calcular os blocos visíveis. Podemos criar hackers de mapas para qualquer jogo de estratégia usando os seguintes métodos:

1. Pesquise por valores desconhecidos.
2. Mova uma unidade para exibir parte do mapa.
3. Filtre o valor alterado.
4. Mova uma unidade para esconder as partes expostas.
5. Filtre o valor alterado.
6. Repita esse processo até que você tenha uma quantidade razoável de resultados (~50).
7. Procure padrões nos resultados e edite cada padrão até descobrir quais padrões representam dados de pavimentação.

Depois de encontrar os dados do bloco, você pode definir pontos de interrupção em um dos blocos. Em seguida, você pode mover uma unidade e um ponto de interrupção aparecerá na função responsável por gravar valores nos dados mapeados.

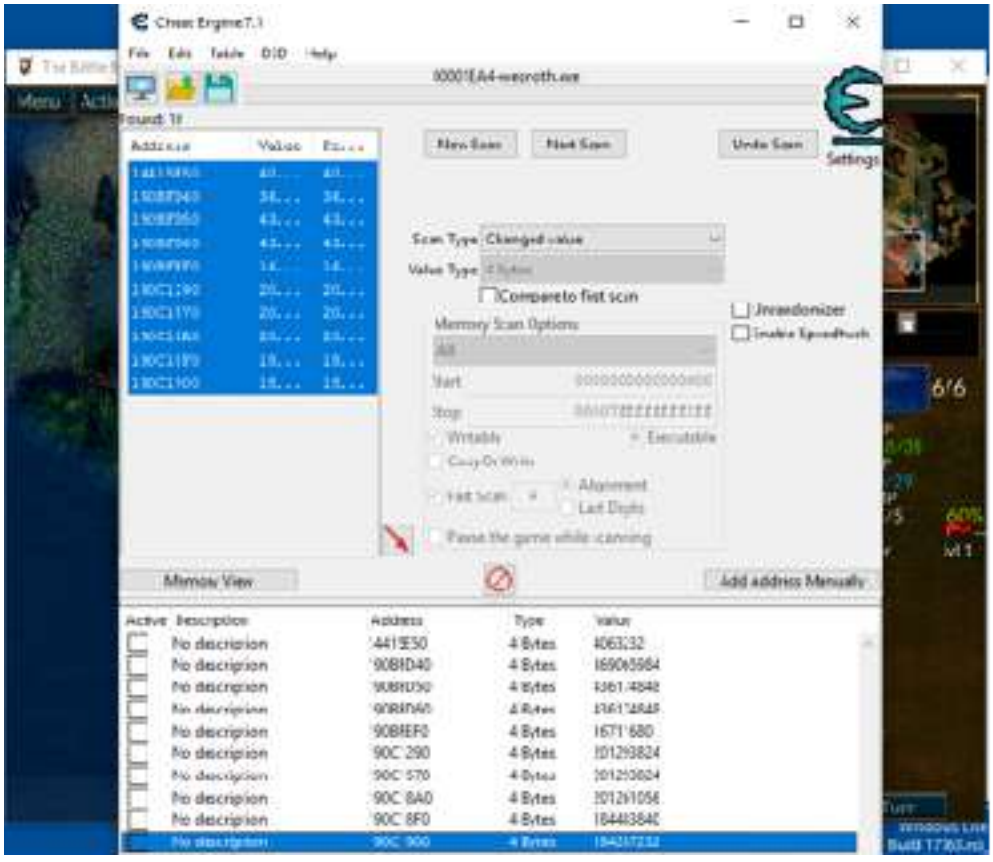
4.2.4 Dados do mapa de localização

Localizar dados de mapas é a parte mais demorada da criação de um hacker de mapas. Primeiro, crie um jogo local no Wesnoth onde um jogador controla seu oponente. Como não sabíamos quais valores pesquisar, começamos a realizar uma nova varredura de tipos de valores desconhecidos.

Após a digitalização, selecione uma unidade e mova-a para um novo local para exibir outros blocos. Certifique-se de se lembrar deste local para que ele possa ser usado em todas as solicitações futuras. Após mover o dispositivo, mude o tipo de varredura para o valor alterado e filtre o resultado . Após a filtragem ser concluída, mova a unidade de volta para a posição inicial, encerre a rodada e esconda o terreno novamente. Encerre rapidamente a rodada do próximo jogador e, em seguida, escaneie o valor alterado novamente quando você recuperar o controle do primeiro jogador. Continue com esse processo até que você filtre os resultados em quantidades gerenciáveis.

Para reduzir rapidamente o número de resultados, você também pode alterar o tipo de varredura para eliminar valores que podem mudar constantemente, mas não estão relacionados ao bloco do mapa. Por exemplo, se você recrutar uma unidade que não exibe blocos adicionais e depois pesquisar por valores constantes, muitos resultados serão filtrados. Essa abordagem pode ser usada em conjunto com diferentes condições (por exemplo, pausar e restaurar o jogo e pesquisar valores inalterados) para reduzir rapidamente o tamanho da pesquisa.

No final das contas, você reduzirá seu conjunto de resultados a um nível razoável, permitindo que você observe manualmente o comportamento do jogo. Neste capítulo, conseguimos reduzir o conjunto de resultados para 10 endereços possíveis. Por motivos do DMA, esses endereços são diferentes toda vez que iniciamos um novo jogo:

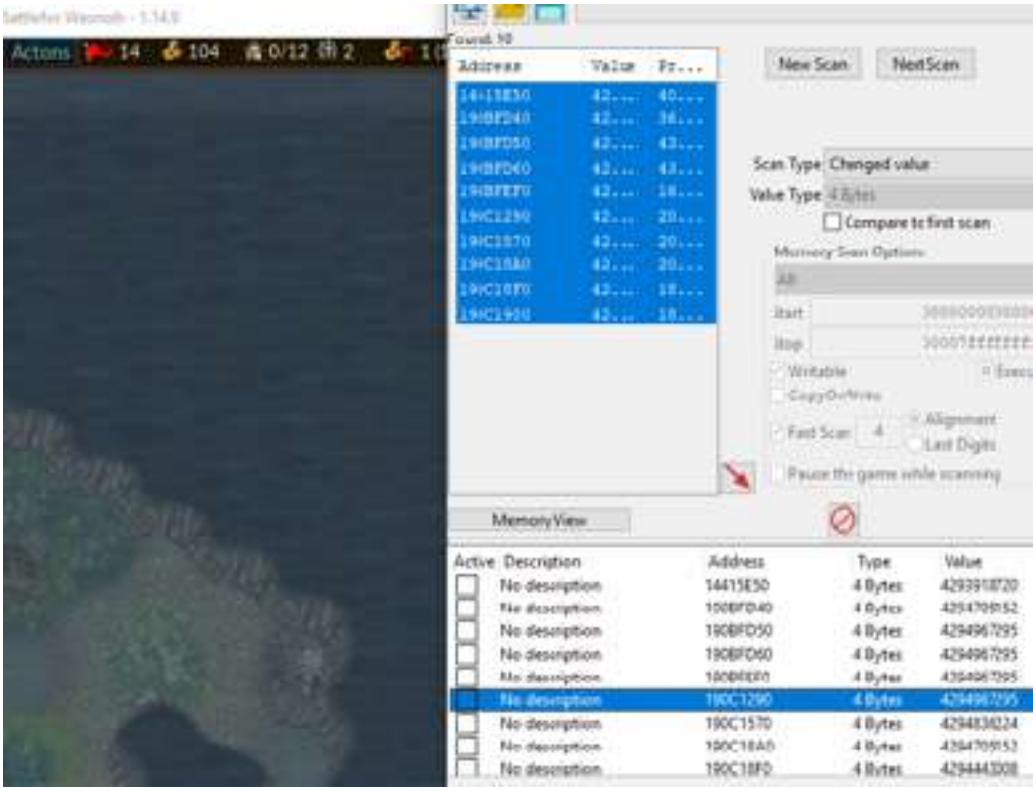


Inicialmente, esses valores pareciam incorretos porque pareciam quase aleatórios. Para determinar se temos o endereço correto, selecione a caixa ao lado de cada endereço na coluna de atividades. Selecionar esta caixa desabilita a modificação do valor do endereço. Se o endereço estiver desativado, remova seu dispositivo da fita que você estava testando. Você deve notar que, ao se mover, os azulejos não exibem mais névoa de guerra. Este teste confirmou que encontramos o endereço correto.

4.2.5 Código do mapa de localização

Agora precisamos determinar como o jogo lida com dados de azulejos de mapas. Ao reverter um jogo desconhecido, começamos fazendo suposições bem fundamentadas. Neste caso, imaginamos que o jogo usa um esquema simples visível/invisível para armazenar blocos individuais para determinar a visibilidade. Usamos esse modelo para ajudar a rastrear dados que nos interessam, mas agora percebemos que ele está incompleto . Antes de prosseguir, precisamos atualizar nossos modelos para refletir nossas descobertas.

Observe o mecanismo de trapaça e mova uma unidade para exibir e esconder azulejos. Você deve observar que, à medida que fazemos isso, os valores que descobrimos mudam constantemente. Quando um bloco está oculto, ele parece ter vários valores grandes. Entretanto, quando o bloco é visível, ele sempre parece ser definido para 4294967295:



Este é um valor diferente, então vamos tentar definir outro endereço que determinamos para esse valor. Após a conclusão, uma coluna inteira de blocos deve ser visível:

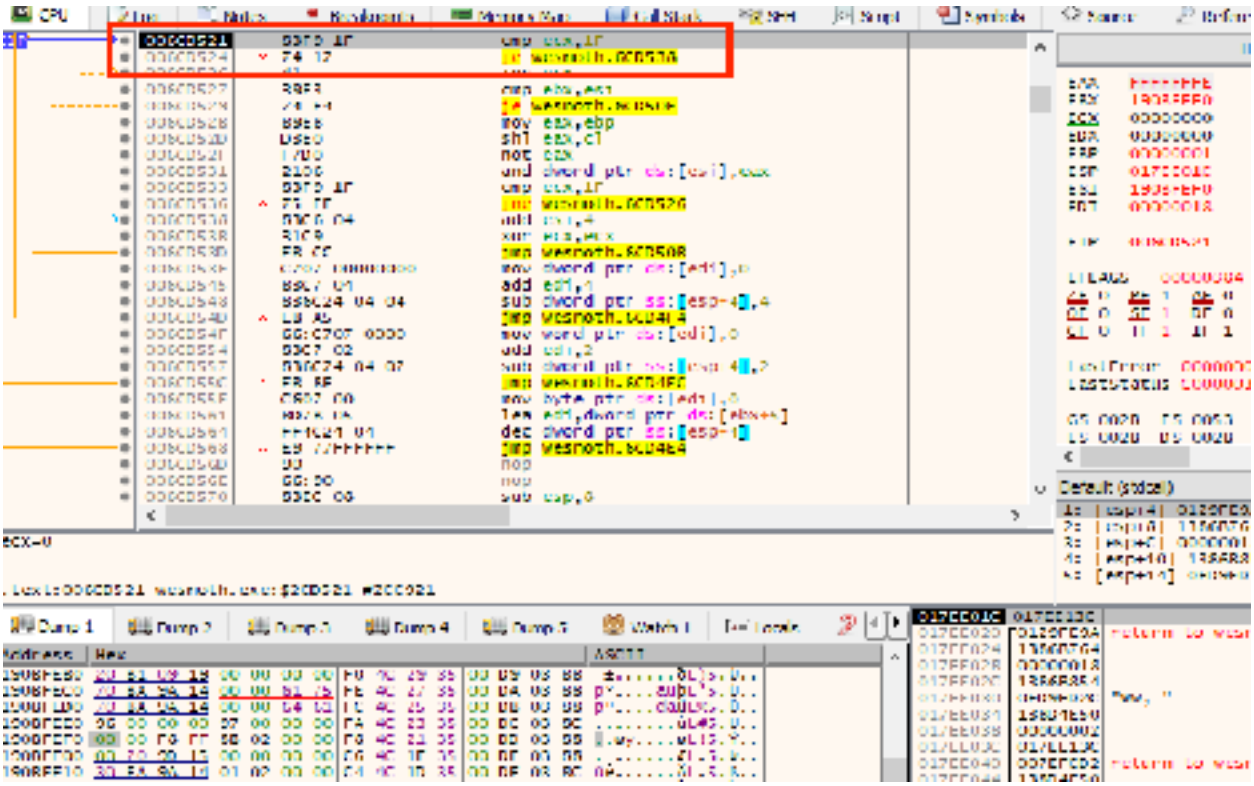
Ao rolar o mapa para cima, você pode ver que toda a coluna do topo do mapa para baixo é controlada pelo seguinte valor:

Com essas informações, agora podemos mudar nossos modelos. Wesnoth parece usar um conjunto de colunas de blocos em vez de um conjunto de blocos. Em seguida, defina essas colunas para valores entre 0 e 4294967295, dependendo do número de blocos visíveis na coluna:

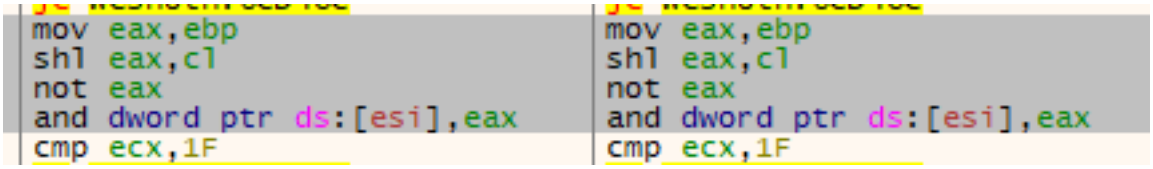
```
map [COLUMN_tamanho]={0,4294967295,...}
```

O valor convertido em hexadecimal de 4294967295 é 0xFFFFFFFF.

Agora podemos começar a rastrear o código responsável por definir esses valores. Adicione x64dbg e defina pontos de interrupção de gravação em um dos endereços das colunas de mapeamento. Mova seu dispositivo para exibir o pavimento e o ponto de interrupção deverá aparecer:



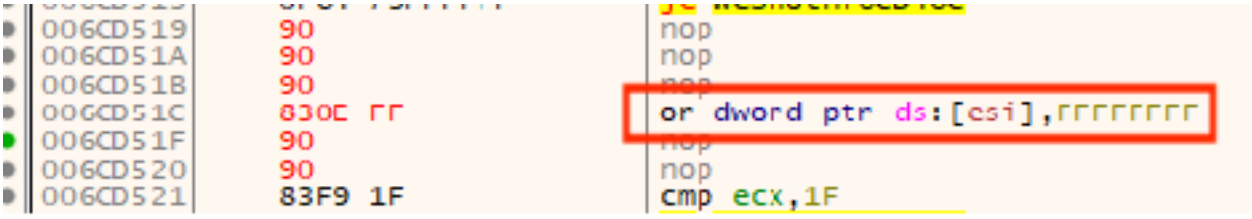
Ao visualizar o registro, podemos reconhecer o esi para salvar nossos dados de colunas de mapeamento . Ao rolar para cima, vemos imediatamente o código responsável por definir o valor desta coluna:



Se verificarmos esse código, veremos um valor carregado no registro eax, modificado e usado para definir o valor da coluna.

4.2.6 Alteração do código do mapa

Como sempre queremos que as colunas pareçam visíveis, podemos modificar esse código e definir o valor da coluna para 0xFFFFFFFF. Faremos isso usando ou operando. A operação or usa dois conjuntos de bits e cria um novo conjunto, onde se ambos os conjuntos de fontes forem 0, o valor de cada bit é 0 e se qualquer conjunto de fontes for 1, o valor de cada bit é 1. Como 0 XFFFFFFFFFFF é convertido em todos os 1, usar esse valor para executar o valor ou sempre gerará 0XFFFFFFFFF. Executaremos isso em nossa coluna de azulejos e emitiremos outras instruções:



Também podemos usar o comando mov para atingir o mesmo objetivo aqui. Entretanto, uma desvantagem das operações mov é seu tamanho ou o número de códigos de operação que precisamos. Só não temos espaço suficiente, precisamos de uma caverna de código. Para evitar essa complexidade adicional, usamos instruções ou porque elas são mais curtas.

Após concluir esta modificação, retorne a Wesnoth e observe que o mapa inteiro agora está visível:

Podemos usar o método semelhante descrito no Capítulo 3.4 para criar um DLL para completar esse comportamento. Primeiro, anote e copie o código de operação gerado pelo x64dbg ao fazer modificações. Colocaremos esses valores em um array para que possamos iterá-los:

```
unsigned char new_bytes[8]={0x90,0x90,0x90,0x83,0x0E, 0xFF,0x90,0x90};
```

Em seguida, como fizemos em outros capítulos, removeremos a proteção da memória na posição do gancho. Em seguida, passamos por cada código de operação na variável new_bytes e o escrevemos na memória do jogo:

```
unsigned char*Hook_Locação=(unsigned char*)0x6CD519; Se(
fdwReason==DL_Process_Attach){
```

```
Projeto Virtualítico ((void*)Hook_Locasionismo, 8. Págão_Exploração,
&ol_Projeto);
for(int i=0; i<sizeof(new_bits); i++){*(hook_
localização+i)=new_bits[i];
}}
```

Então você pode injetar esse DLL como fizemos em todos os capítulos anteriores. Quando injetado, nosso hacker de mapas exibirá os azulejos de cada mapa do jogo.

O código completo para este capítulo está disponível no Apêndice A.

4.3 Macro Robôs

4.3.1 Objetivos

Neste capítulo, mudaremos nosso foco para o jogo Wyrmsun, versão 5.0.1. Isso ocorre porque nosso objetivo atual, o Wesnoth, não suporta mecânicas de jogos que nos permitem escrever macrorobôs (como controle em tempo real de unidades de construção). Wyrmsun é gratuito, semelhante a outros jogos RTS tradicionais, como StarCraft, Warcraft ou Command & Conquer.

4.3.2 Identificação

Macrobot é um hacker que monitora nossos recursos e constrói unidades de trabalho automaticamente. Neste capítulo, criaremos um macrobot que, quando nosso dinheiro ultrapassar 3.000, construirá automaticamente um trabalhador da estrutura atualmente selecionada.

4.3.3 Compreensão

Para escrever um macrobot, precisamos entender como os jogos RTS lidam com a criação de unidades. Normalmente, os jogos RTS têm uma lista de unidades associadas a cada jogador. Ao criar uma unidade, o jogo executa várias ações e depois adiciona uma nova unidade à lista. O código pode ser o seguinte:

```
recruit_unit(Unit_Tipo){
memory=Initalize_Memory (tamanho(unit))unit=
Creat_Unit(Unit_Tipo, memory)
Player->DeCerease_Money()
Player->add_unit (unidade)
player->Increase_POPULATIO_Conter()}
```

Para criar uma unidade, precisamos encontrar essa função e chamá-la nós mesmos. Para encontrar essa função, podemos usar duas abordagens diferentes, dependendo de como o jogo lida com a criação de unidades.

- Se criarmos uma unidade e o jogo reduzir imediatamente nosso dinheiro, posicionaremos nosso dinheiro e definiremos um ponto de interrupção no valor do nosso dinheiro.
- Se criarmos uma unidade, o jogo imediatamente aumenta nossa população, localizaremos nossa população e definiremos um ponto de interrupção de escrita em nossa população atual.

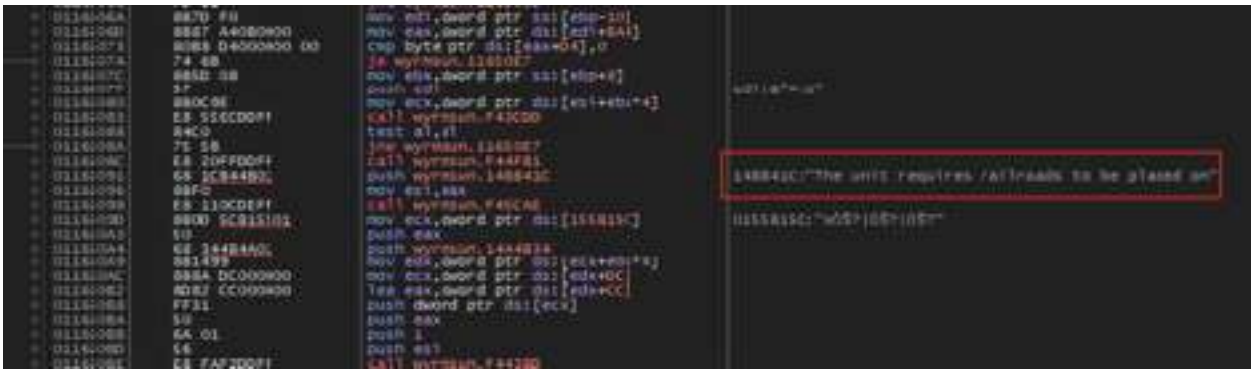
Quando esses pontos de interrupção são acionados, nós os colocaremos na função decrease_money ou Increase_PoUTAN_Conter presumida no código de exemplo acima. Então precisamos adicionar alguns recursos. Faremos isso executando a função até que ela retorne e depois sair.

Wyrmsun (objetivo deste capítulo) carrega código dinamicamente. Portanto, os endereços que você vê neste capítulo serão diferentes quando você segue. Entretanto, as instruções e métodos descritos não serão alterados. Discutiremos como lidar com esse comportamento ao criar um DLL.

4.3.4 Localização e criação da função da unidade

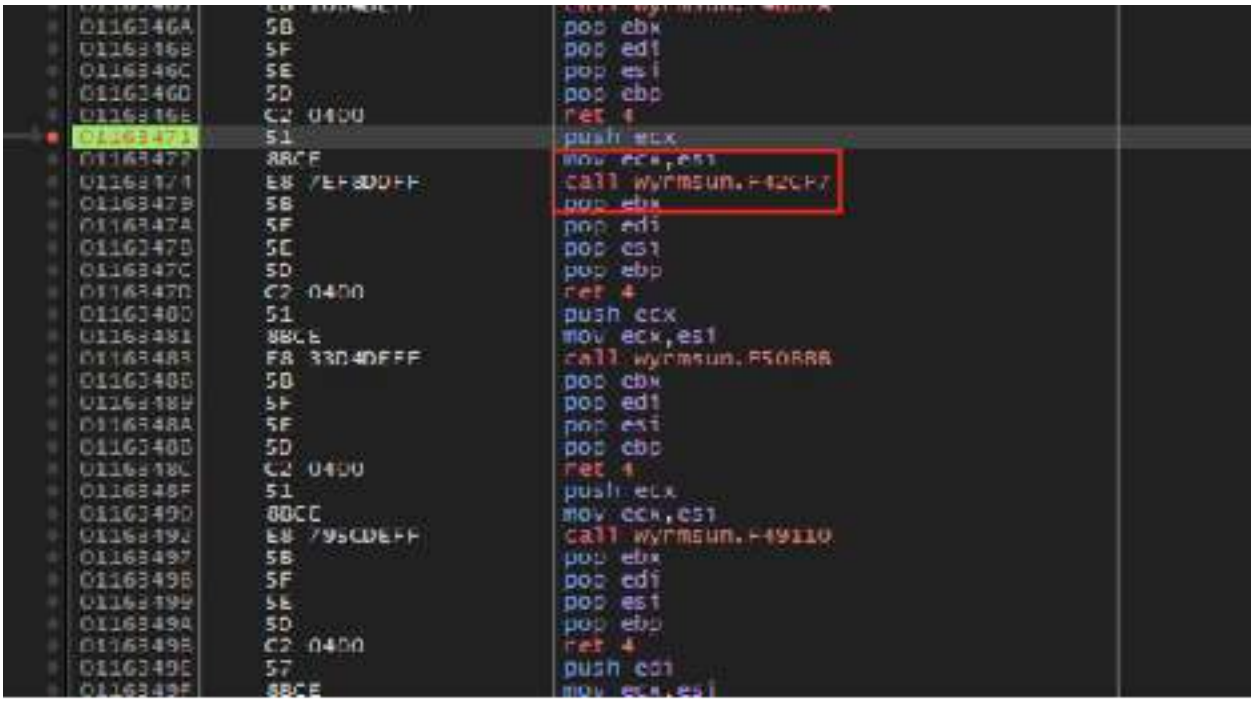
Em Wyrmsun, quando você recruta uma unidade, o dinheiro diminui imediatamente, então usaremos o primeiro método mencionado na seção anterior. Para encontrar o endereço de financiamento para esse novo alvo, podemos usar a abordagem discutida no Capítulo 1.5.

Em seguida, adicione x64dbg ao jogo. Certifique-se de que nenhuma operação que possa alterar seus fundos ocorrer e defina pontos de interrupção de hardware no endereço do fundo durante a gravação. Ao recrutar um trabalhador, o ponto de interrupção deve aparecer imediatamente. Usando o Excesse until Retour/step over, execute vários níveis de código para cima até ver a seguinte string:

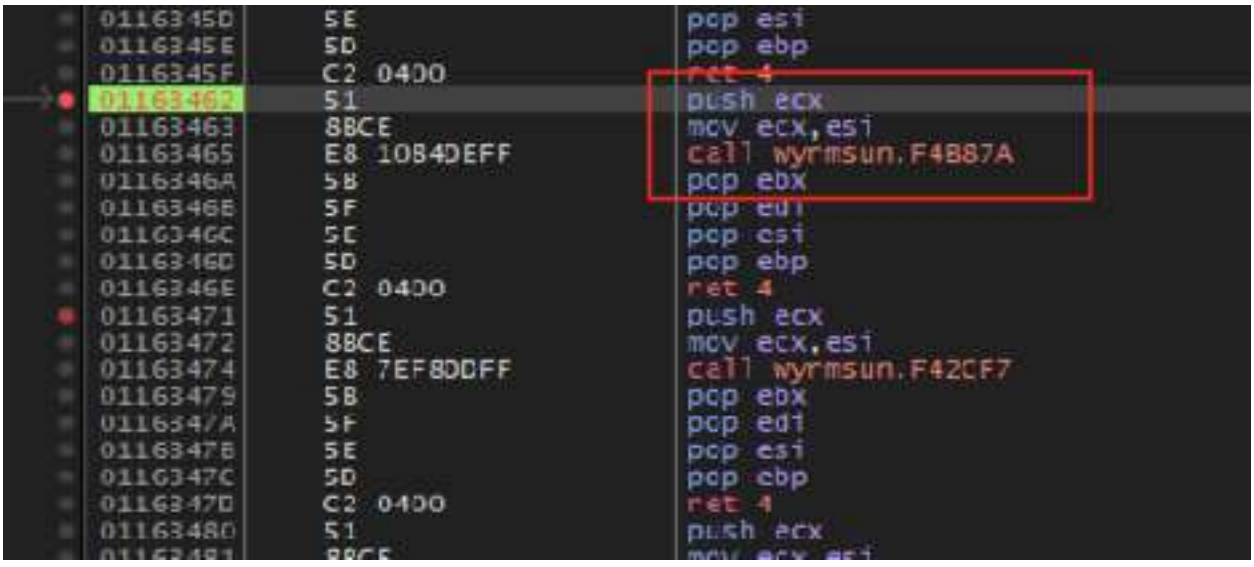


A partir dessa sequência, podemos ver que estamos no lugar certo, porque essa lógica está claramente relacionada às unidades de recrutamento e colocação.

A partir daqui, navegue mais um nível até a função de chamada pai:



Se verificarmos esse código, veremos que é uma série de chamadas muito semelhantes, aceitando apenas um parâmetro. Se definirmos um ponto de interrupção na chamada de 0xF42CF7, podemos ver que uma unidade só é chamada quando ela é recrutada. Em seguida, coloque um ponto de interrupção na chamada acima da chamada 0xF42CF7. Com esse conjunto, você pode realizar várias ações no jogo, como mover unidades, atacar e construir. Ao executar a operação de construção, o ponto de interrupção da chamada acima da função de recrutamento da unidade deve ser exibido:



Dado esse comportamento, podemos especular que todas essas chamadas estão relacionadas a funções na placa de célula (canto inferior direito da tela). Podemos imaginar que o código poderia ser o seguinte:

```
Interruptor(menu_event){
Construção de caso:
build_structure (Evente_Data);
descanso;
Recrutamento de caso:
recruit_unit(event_data); descanso;

Movimento de caso:
move_unit(event_data); descanso;

}
```

Portanto, podemos presumir que a chamada 0xF42CF7 (em nosso exemplo) seja responsável pela unidade de recrutamento. Podemos verificar isso omitindo as seguintes instruções:

```
Impulsionar ecx
mov ecx, chamada
esi 0xF42CF7
```

Ao notar, clicar no botão recruit na estrutura não cria mais a unidade.

4.3.5 Estrutura de dados de eventos reversos

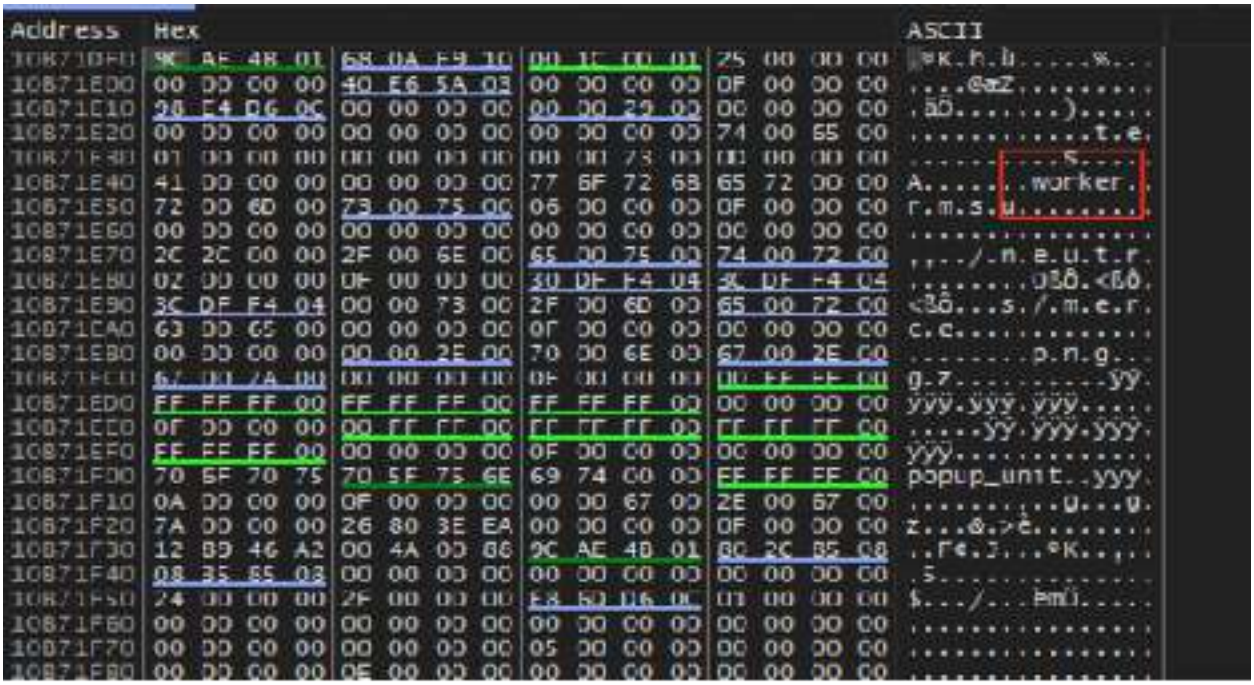
Agora que encontramos a função de criação de células, precisamos reverter os dados fornecidos a ela para que possamos chamá-la nós mesmos. Podemos ver que há dois dados que podem ser transferidos para uma função:

- Um valor em ecx, que é empurrado para a pilha
- Valor em esi, o valor é movido para ecx

Vamos determinar se ambos são necessários. Primeiro, substitua o comando push ecx por outro registro, como push eax. Se você tentar recrutar uma unidade no jogo, notará que o jogo trava imediatamente, o que sugere que o push é importante. Em seguida, reinicie o jogo e saia do mov ecx e do comando esi. Você notará que o jogo está respondendo normalmente, indicando que essa operação não está sendo usada pelo evento de criação da unidade. Portanto, precisamos apenas reverter o valor do ecx durante o push.

Defina pontos de interrupção no push ecx e recruta um trabalhador. Quando o ponto de interrupção aparecer, clique com o botão direito no valor ecx e selecione Follow in dump:

Esses dados não parecem conter todas as informações que esperávamos. O valor de similaridade dos dados (as três primeiras entradas são repetidas no final de 0xb710) indica que este pode ser um ponteiro em uma tabela de ponteiros. Para verificar essa hipótese, selecione 4 bytes (0xf01db710) e selecione Follow in dump. Sua visualização de dump deve ser alterada para 0x10b71df0, conforme mostrado abaixo:



Os escritores de texto devem pular na sua frente imediatamente. Essa estrutura parece conter dados sobre a unidade a ser criada. De volta ao jogo, crie outra estrutura para criar uma unidade (como um salão de guerra ou um edifício do tipo quartel) e uma unidade de infantaria. Quando o ponto de interrupção aparecer, verifique a parte da memória em 0x10b71df0 novamente:

Podemos ver mudanças em dois dados principais entre trabalhadores e unidades de infantaria recrutadas: texto (trabalhadores contra infantaria) e números pré-numerados (0x41 pares 0x2c). Podemos assumir que esse número pode ser uma representação interna do tipo de unidade.

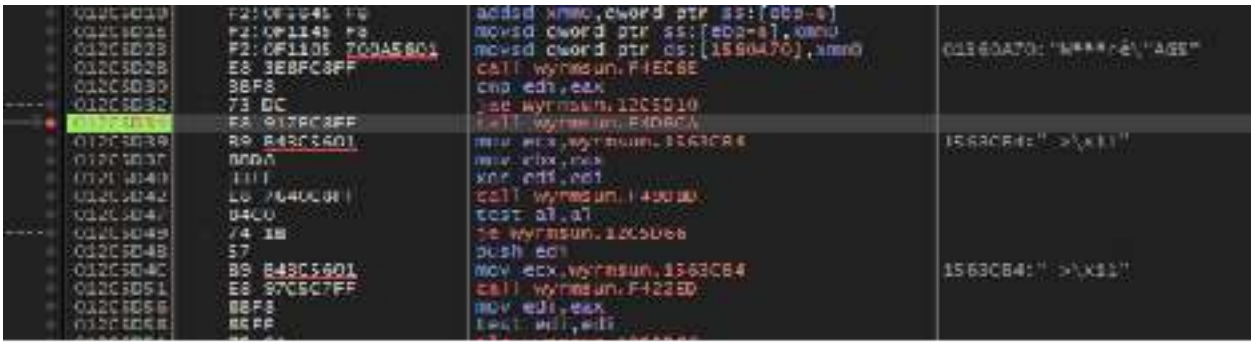
Vamos verificar se isso funciona porque ainda estamos especulando. Primeiro, defina um ponto de interrupção no comando ecx push para criar uma unidade de infantaria. Em seguida, quando o ponto de interrupção aparecer, mude 0x2C para 0x41 e mude a infantaria para trabalhadores. Se você continuar executando e voltar ao jogo, verá que, apesar de clicar em infantaria, agora estamos criando um trabalhador da estrutura do quartel.

Há outros dados na estrutura que mudam, mas não estão diretamente vinculados à célula que está sendo criada. Embora possamos reverter toda a estrutura, copiaremos a estrutura quando criarmos o worker e usaremos esses valores para nossos ataques de hackers.

4.3.6 Posicionamento do loop principal do jogo

Agora que encontramos a função responsável pela unidade de recrutamento e sabemos como chamá-la, precisamos encontrar um lugar para chamá-la. Para este capítulo, escolheremos ganhar o loop do jogo principal. No nosso exemplo, encontrar o loop principal do jogo é fácil. Primeiro, coloque um ponto de interrupção na chamada para recrutar uma unidade e crie uma unidade no jogo para que

O ponto de interrupção aparecerá. Em seguida, continue avançando em cada função. No final, você receberá as seguintes ligações:

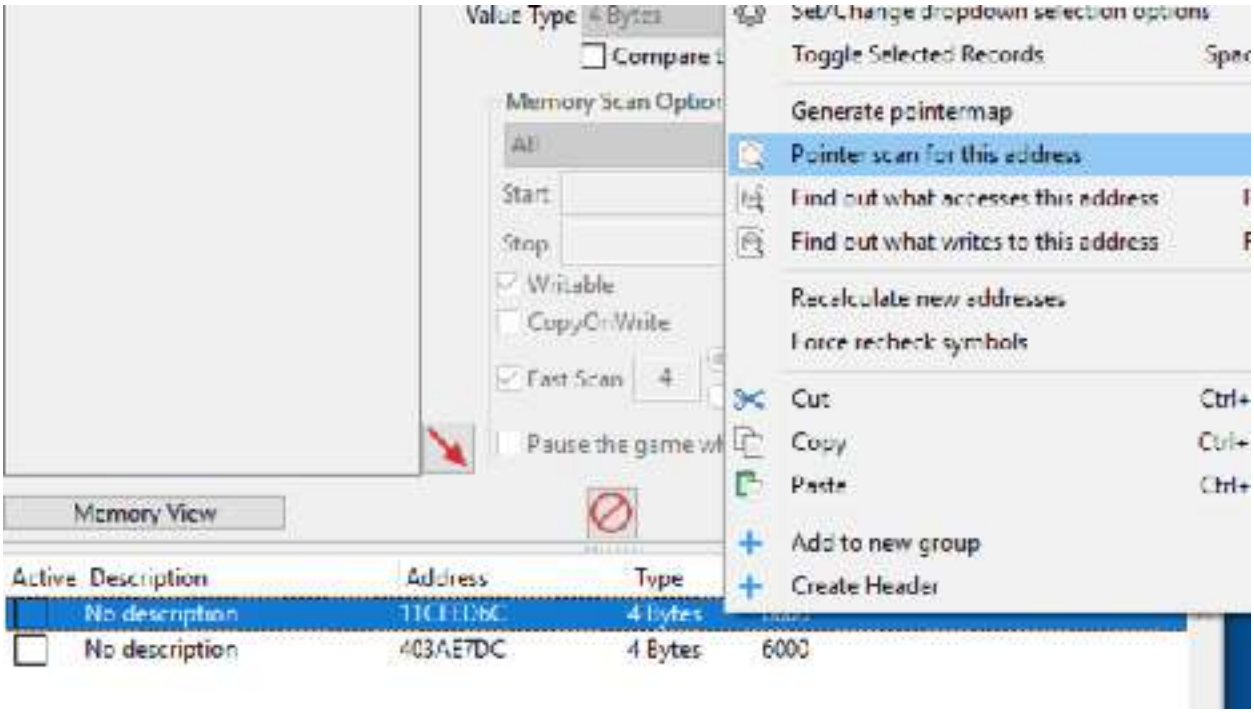


Se você tentar executar até retornar aqui, notará que o jogo começará e continuará executando. Isso ocorre porque estamos em um ciclo e não encontramos instruções ret. Podemos verificar esse comportamento definindo pontos de interrupção nesta chamada. Você notará que pontos de interrupção continuam aparecendo. Ambos os fatores sugerem que o código faz parte do loop principal do jogo.

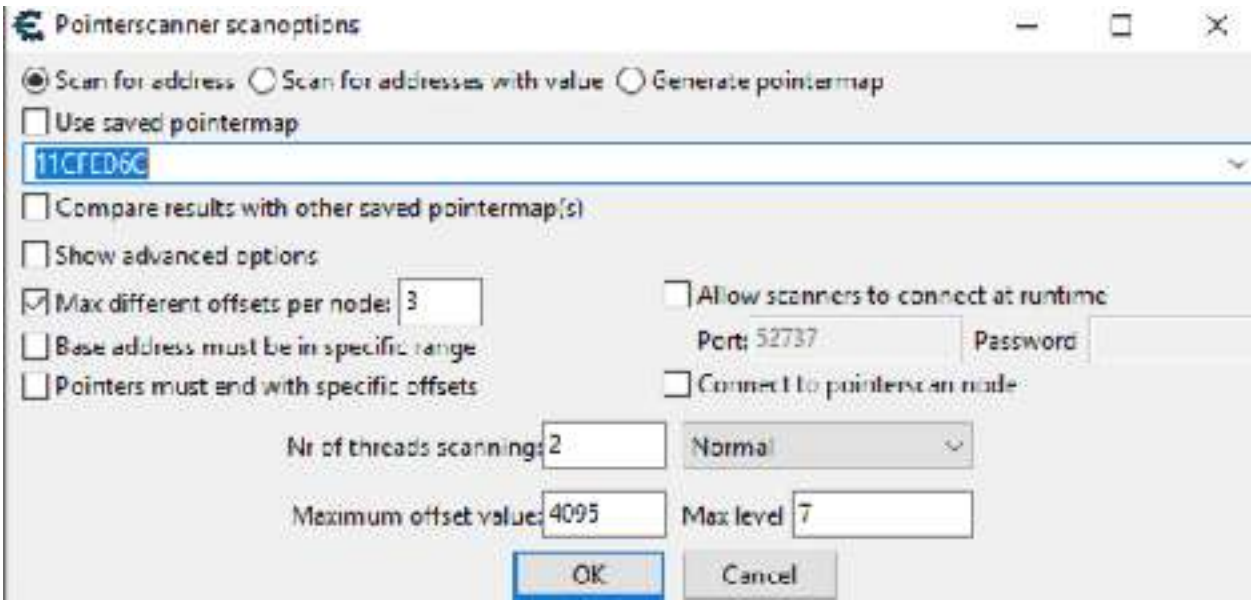
4.3.7 Posicionando o dinheiro dos jogadores

Por fim, queremos monitorar o dinheiro dos nossos jogadores para nossos hackers. Assim como em outros jogos, o Wyrmsun aloca dinheiro dinamicamente aos jogadores, o que significa que o dinheiro para cada jogo é diferente. Em capítulos anteriores, discutimos maneiras de derrotar o DMA. Neste capítulo, usaremos o recurso de varredura de ponteiro do mecanismo de trapaça em vez de reverter o alvo.

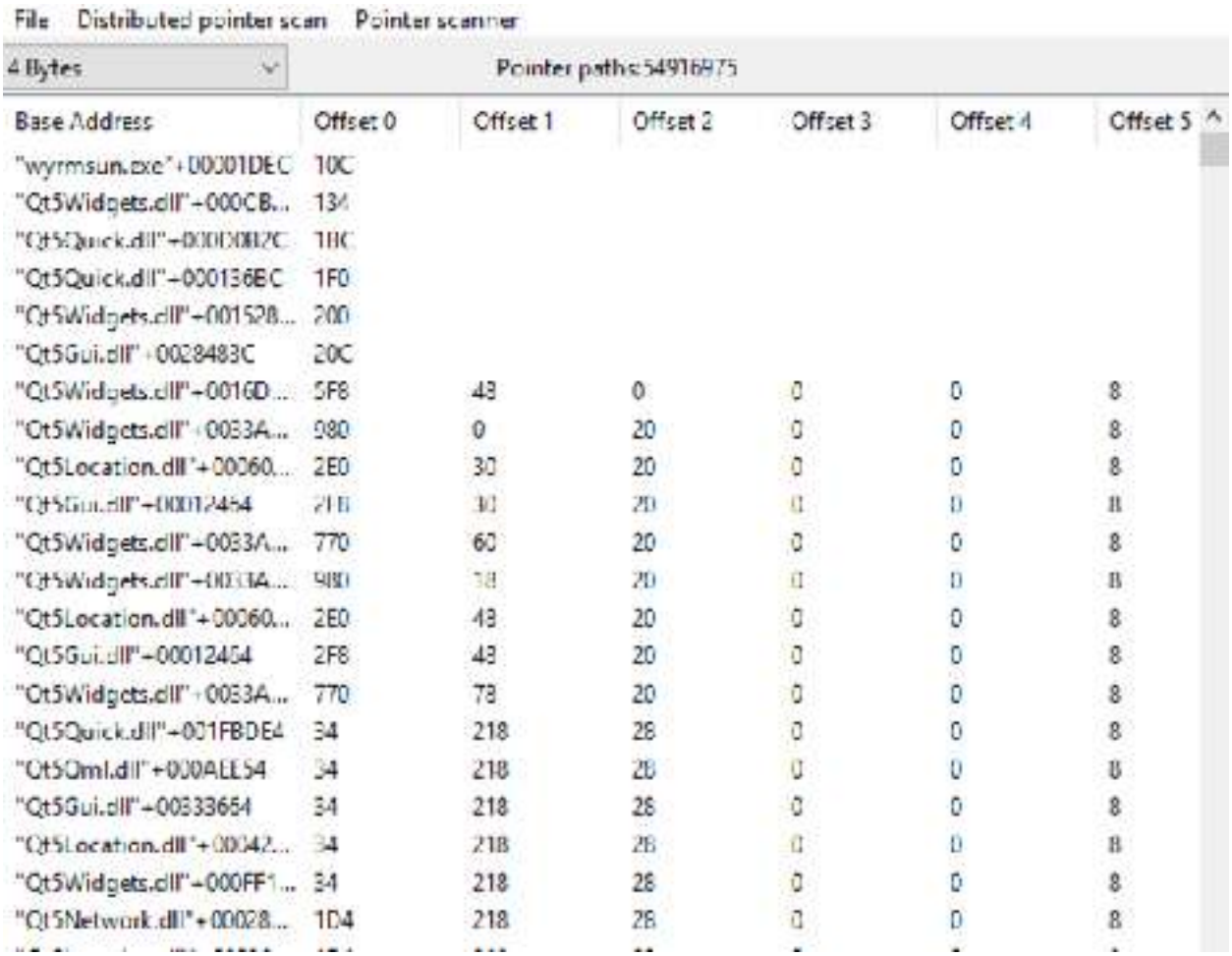
A varredura de ponteiros do mecanismo fraudulento funciona de forma semelhante à varredura de memória Primeiro, precisamos encontrar nosso endereço financeiro como de costume. Em seguida, clique com o botão direito no endereço e selecione o ponteiro para



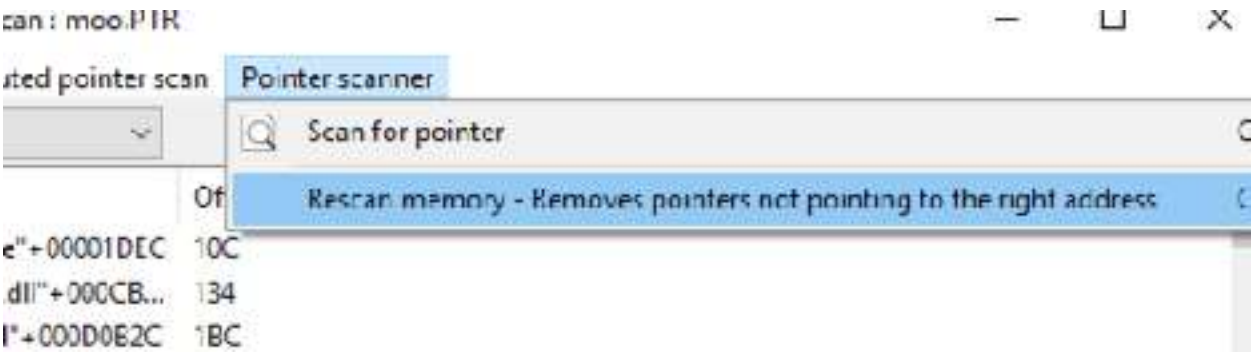
Na caixa de diálogo que aparece, guarde todas as opções padrão e selecione OK. Quando o prompt aparecer, selecione o arquivo em qualquer lugar:



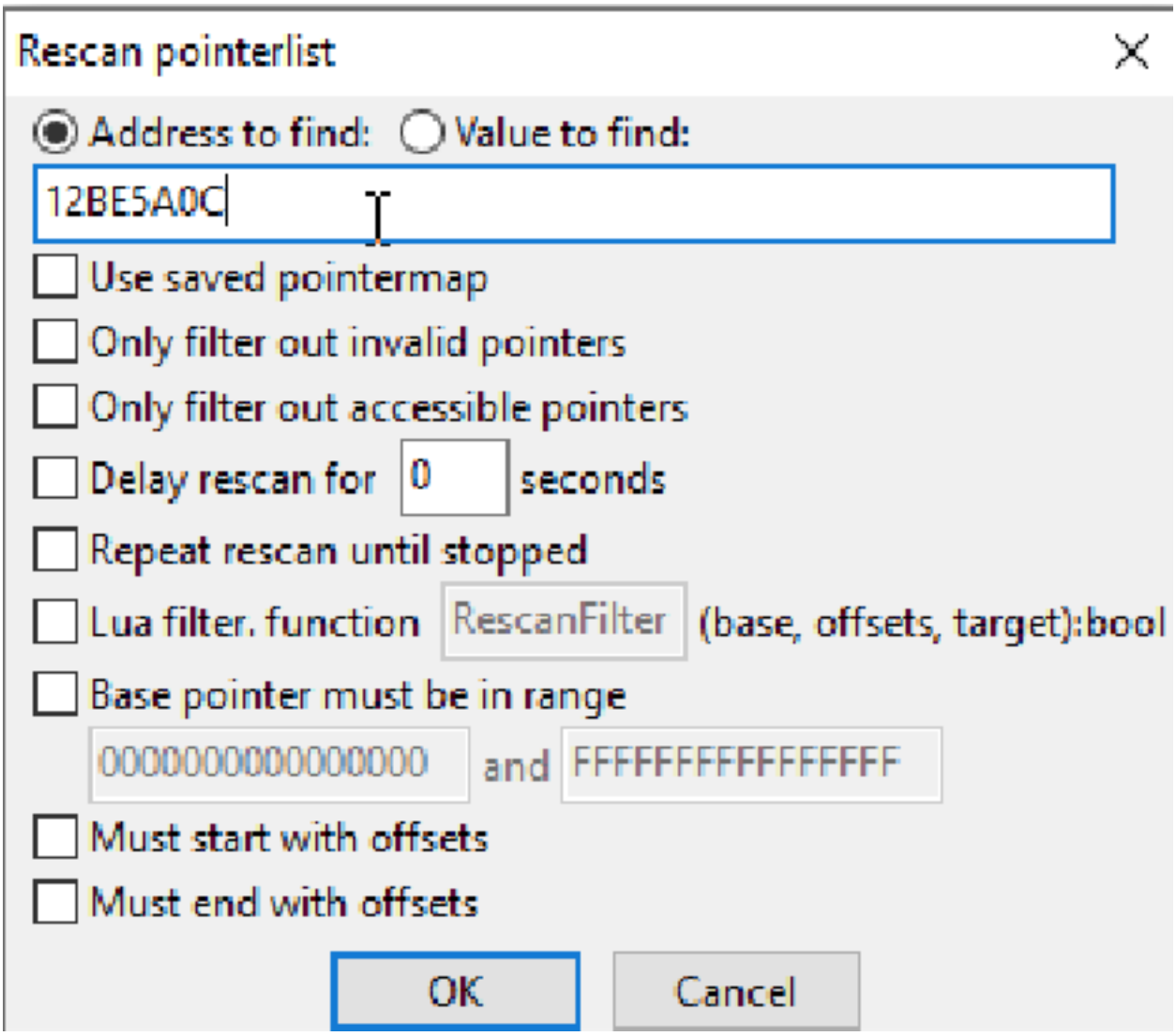
O mecanismo de trapaça agora pesquisará o alvo e apontará para o selecionado Resolva de alguma forma. Quando concluído, você retornará milhares de resultados:



Assim como nas digitalizações regulares, agora precisamos filtrar esses endereços. Recomece sua corrida para que seu alvo possa ser movido para um novo endereço. Em seguida, encontre seu endereço financeiro novamente. Em seguida, na janela de digitalização do ponteiro, selecione reescanear a memória



Na caixa de diálogo que aparece, insira seu novo endereço e clique em OK:



Assim como na filtragem convencional, o mecanismo de fraude agora reescaneará todos os ponteiros reconhecidos anteriormente para ver se eles ainda apontam corretamente para seu novo endereço. Repita essa operação várias vezes e você descobrirá que alguns ponteiros sempre apontam corretamente para o valor financeiro do jogador. Neste capítulo, usaremos:

```
+0x14
[+0]
[+0x4]
[+0x8]
[+0x4]
[+0x78]
wyrmsun.exe+0x0061A504
```

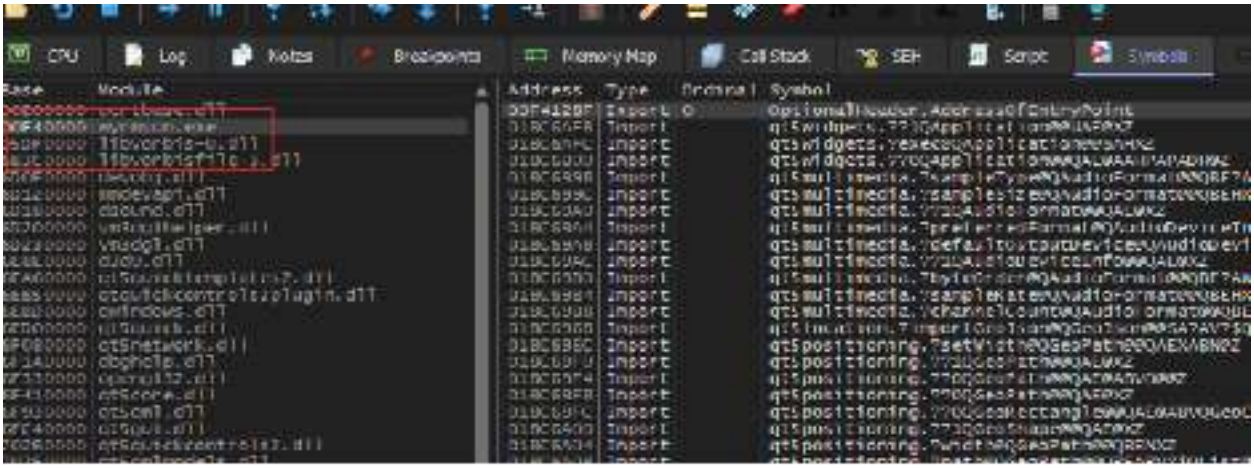
Discutiremos como usar esses valores em seu código, então substitua qualquer valor que você encontrar.

4.3.8 Processamento de código dinâmico

No início deste capítulo, discutimos como o código é carregado dinamicamente para que os endereços não sejam consistentes. Você pode verificar esse comportamento iniciando o Wyrmsun, anotando o endereço, reiniciando a máquina virtual e ativando o Wyrmsun novamente. Você notará que o endereço não contém mais o mesmo código.

Assim como o DMA, sabemos que um jogo deve ter alguma maneira de localizar seu código. Ao processar código dinâmico, o jogo normalmente desloca todos os endereços do endereço base do módulo do jogo. Podemos observar esse comportamento observando a chamada da unidade de criação. Embora o primeiro byte mude, a chamada sempre termina em 0x2CF7 (por exemplo, 0xf42c7 ou 0x292c7).

Podemos determinar o endereço do módulo principal acessando a aba Symbols em x 64dbg:



Podemos ver aqui que nosso local é 0x00F40000. Portanto, sabemos que a função da célula de criação existirá no endereço +0x2CF7. Da mesma forma, vemos neste capítulo que a chamada para a unidade de criação está localizada em 0x01163471. Se subtrairmos esse endereço do endereço base, obtemos um deslocamento de 0x223471. Podemos usar esses deslocamentos para calcular automaticamente o endereço da função com a qual nos interessamos ao criar DLL.

4.3.9 Crie nosso DLL

Assim como nos capítulos anteriores, criaremos um DLL para injetá-lo em nossos objetivos. Primeiro, começaremos com nossos fundamentos:

```
#include<Windows.h>

BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      DWORD old_protect;

      if(fdwReason==DLL_PROCESS_ATTACH){//
Código está vinculado aqui
}

Retornar a
verdade;}
```

Nosso DLL terá duas cavernas de código. Primeiro, criaremos uma caverna de código no código responsável pela criação da unidade. Nesta caverna de código, recuperaremos o valor do ecx e copiaremos a estrutura à qual ele aponta para a memória do DLL. Nossa segunda caverna de código envolverá o loop do jogo principal. Nesta caverna de código, verificaremos o valor da moeda do jogador atual e, em seguida, chamaremos a função create unit.

4.3.10 Criação de Caverna de Código de Unidade

Quando invertemos a função create unit, identificamos a estrutura enviada para a função como parâmetros. Embora tenhamos identificado vários componentes dessa estrutura, não a revertemos completamente. Como essa estrutura não muda ao criar uma unidade de trabalho, usaremos a caverna de código para copiar a forma efetiva da estrutura. Em seguida, usaremos esse formulário copiado em nosso jogo principal, loop code Cave.

Primeiro, ganhamos o endereço da unidade de criação e o direcionamos para nossa caverna de código, como fizemos nos capítulos anteriores. Como o endereço do código mudou, determinaremos o endereço com base no endereço do módulo. Também usaremos este endereço base para calcular a localização da chamada da unidade de recrutamento:

```
O cabo de frase wyrmsun_base;
```

```
DWORD RET_Unit_RET_Adress;
DWORD Recruit_Unit_Call_Adress;

...

wyrmsun_base=GetModuleHandle (L"wyrmsun.exe");

Card*Hook_Localização=(Card*)(Card*) ((DWORD)wyrmsun_base+
0x223471);
recruit_unit_ret_address=(DWORD)hook_location+8;
Recrut_Unit_Call_Adress=(DWORD)wyrmsun_base+0x2CF7;

Projeto Virtual* (void*)Hook_Locasione, 8. página _Express_Readwrite,
&old_protect);
*Hook_Localização=0xE9;
*(DWORD*)(hook_localização+1)=(DWORD)&recruit_unit_codecave-
((DWORD)hook_localização+5);
*(hook_localização+5)=0x90;
*(hook_localização+6)=0x90;
*(hook_localização+7)=0x90;
```

Em nosso recruit unit code cave, primeiro recuperaremos o valor do ecx e o colocaremos em uma variável:

```
Cardinal DWORD*;

__declspec(naked)void recruit_unit_codecave(){_
asm{
Base pushad mov, ecx}
```

Agora, armazenando este ponteiro na variável base, podemos cancelar a referência deste ponteiro para recuperar a localização da estrutura. Ao cancelar referências com um ponteiro, podemos copiar toda a estrutura para outra variável para uso em outra base de código. Podemos recuperar o tamanho observando o tamanho da estrutura em x64dbg. Além disso, criaremos uma variável init para rastrear se isso já aconteceu:

```
DWORD*unitbase;

Caracteres sem símbolo unitdata[0x
110]; Inicialização booleana = falsa;
```

```
...

unit base=(DWORD*)(*base); memcpy (
unitdata, unitbase, 0x110); init=
verdade;
```

Por fim, restauraremos nossos registros e instruções originais:

```
_asm{
Rebote
Impulsionar ecx
MiECX, esi
Chamar para recruit_Unit_Call_adress
jmp recruit_unit_retr_adress}

}
```

4.3.11 Game Loop Code Cave

Depois de copiar os dados no buffer, agora podemos criar uma caverna de código do loop do jogo. Como antes, começaremos conectando o endereço que determinamos anteriormente:

```
DWORD GameOP_Ret_Adress;
DWORD Gameloop_Call_Adress;

Hook_Localização=(unsigned char*)((DWORD)wyrmsun_base+0x385D34);
gameloop_ret_address=(DWORD)hook_localização+5;
Gameloop_Call_Adress=(DWORD)wyrmsun_base+0xDBCA;

Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
*Hook_Localização=0xE9;
*(DWORD*)(hook_localização+1)=(DWORD)&gamioop_codecave-((
DWORD)hook_localização+5);
```

Em nosso jogo Loop Code Cave, primeiro verificaremos o valor do dinheiro do jogador . Faremos isso usando ponteiros e deslocamentos recebidos do mecanismo de trapaça:

```
DWORD*gold_base, *gold;
```

```
...
__declspec(naked)void gameloop_codecave(){
_asm{
Pushad
}

gold_base=(DWORD*)((DWORD)wyrmsun_base+0x0061A504); gold=(
DWORD*) (*gold_base+0x78);
Ouro =(DWORD*) (*ouro + 4); Ouro =(
DWORD*) (*ouro + 8); Ouro =(DWORD*) (
*ouro + 4); Ouro =(DWORD*) (*ouro);
gold=(DWORD*) (*gold+0x14);
```

Em seguida, verificaremos se nosso buffer unitário foi inicializado e se o dinheiro do jogador ultrapassa 3.000. Se for esse o caso, copiamos o buffer do trabalhador para o buffer apontado pelo jogo e movemos a base para o ecx antes de chamar a função recruit unit:

```
if(init&&*gold>3000){memcpy(unitbase,
unitdata, 0x110); _asm{

mov ecx, base
Impulsionar ecx
Chamar Recruit_Unit_Call_Adress}

}
```

Mais uma vez, precisamos restaurar as instruções originais:

```
_asm{
Rebote
Chamar Gameloop_Call_AdressJMP
Gameloop_Ret_Adress_}

}
```

Por fim, podemos construir esse DLL e injetá-lo em nosso jogo. No jogo, recrutamos uma unidade para copiar nosso buffer e começar a cobrar dinheiro. Você deve notar que os trabalhadores começam a ser recrutados imediatamente.

O código-fonte completo deste capítulo é mostrado no Apêndice A.

Parte 5
Hacker FPS

5.1 Fundamentos 3D

5.1.1 Visão geral

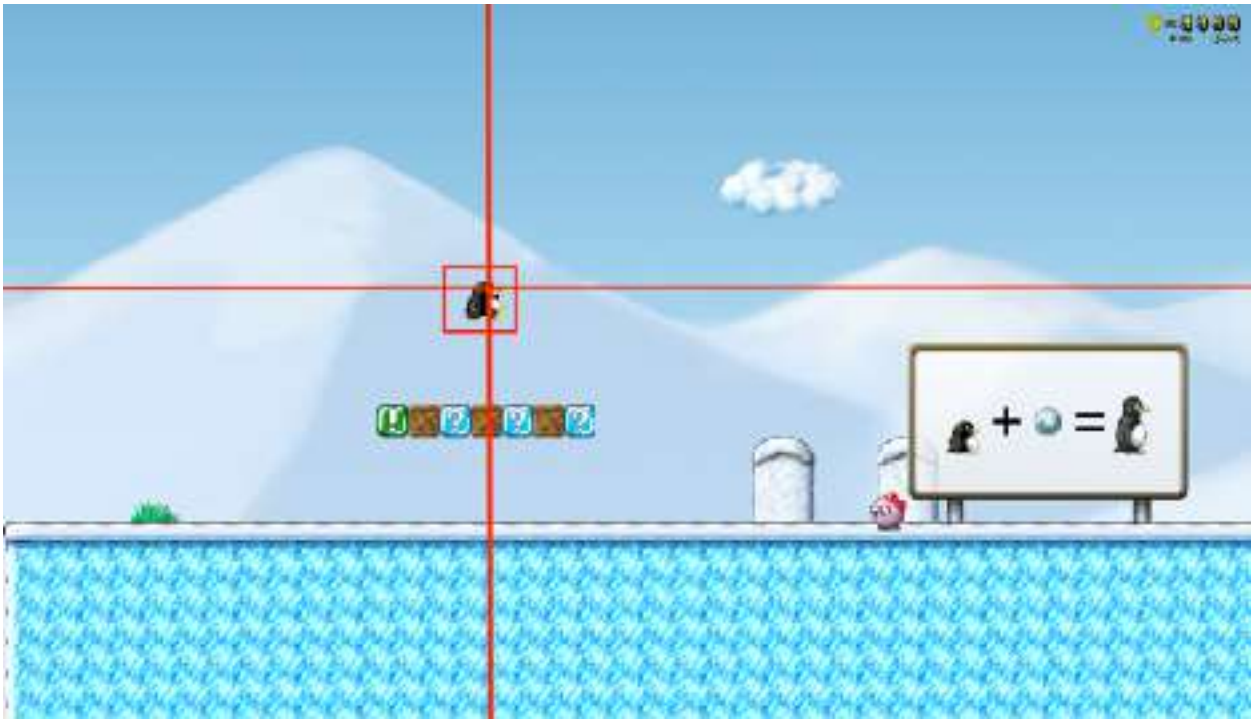
Nos capítulos anteriores, nos concentramos em como decifrar jogos bidimensionais (2 D). Embora muitas das tecnologias que apresentamos possam ser aplicadas a qualquer jogo, existem algumas tecnologias exclusivas que são adequadas apenas para jogos 3 D (3D). Para criar hackers como wallhacks ou aimbots para jogos 3D, precisamos entender como os jogos 3D realmente funcionam.

5.1.2 Coordenadas

Quando dizemos que um jogo é 2D, queremos dizer que todos os objetos do jogo podem ser localizados por meio de um par de coordenadas. Esses pares de coordenadas contêm dois valores: X (posição horizontal) e Y (posição vertical). Pares de coordenadas geralmente são citados entre parênteses, como (X, Y).

Tomando como exemplo a captura de tela do Wesnoth, suponha que haja um ponto (0 ,0) no canto inferior esquerdo da tela e um ponto (10,10) no canto superior direito. As unidades destacadas podem ser representadas por coordenadas (7, 5), e as unidades não destacadas podem ser representadas por coordenadas (3, 1).

O jogo usa essas coordenadas para realizar muitas operações importantes. Por exemplo, quando um jogador tenta se mover, o jogo verifica que as novas coordenadas do jogador não estão na água ou em terrenos inacessíveis. Todos os jogos 2D usam coordenadas dessa maneira, independentemente de o jogo ser de cima para baixo ou de lado.



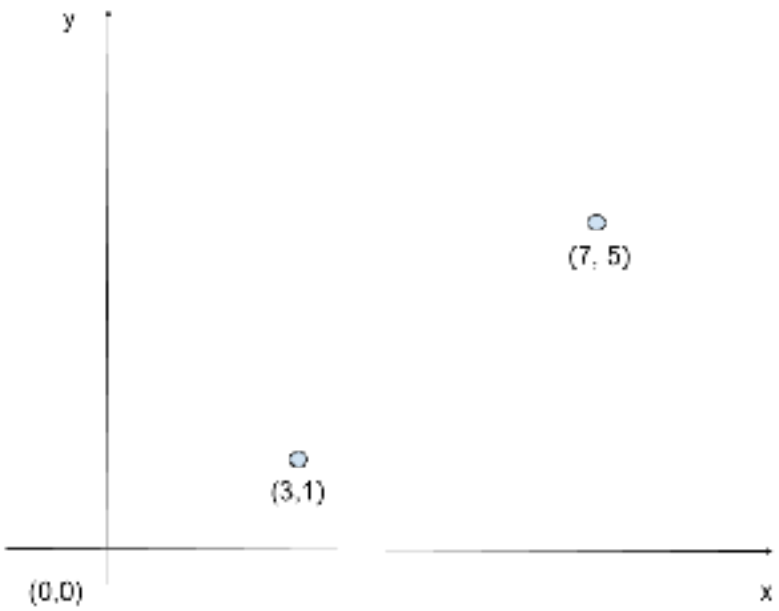
5.1.3 Espaço Tridimensional

Ao jogar Wesnoth, você pode notar uma coisa: duas unidades nunca podem compartilhar as mesmas coordenadas. Isso ocorre porque, sem uma lógica especial para lidar com a alternância entre duas imagens, o jogo não conseguirá exibir cada unidade corretamente ao jogador. Entretanto, em jogos 3D, duas unidades podem compartilhar as mesmas coordenadas horizontais e verticais.

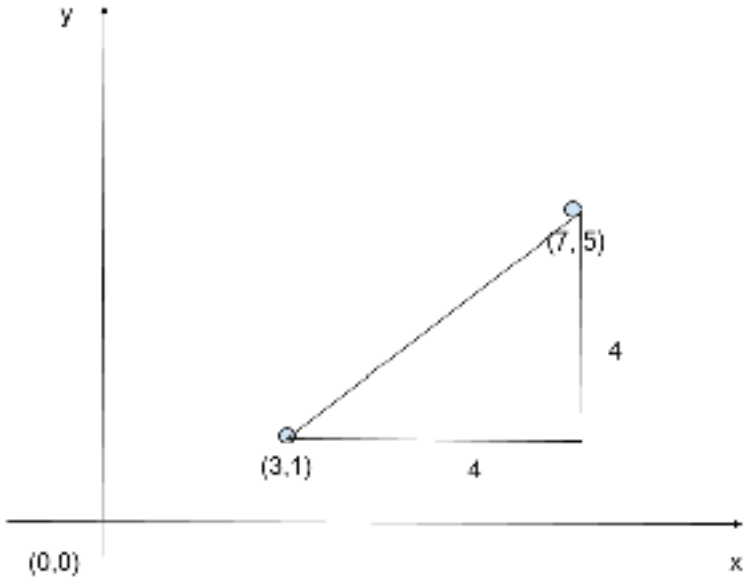
Como você pode ver, outro jogador e nosso jogador atual estão no meio da tela e têm o mesmo valor (X, Y). Entretanto, em jogos 3D, as coordenadas são representadas em três valores: X, Y e Z (profundidade). No exemplo acima, ambos os jogadores podem estar no espaço 2D (5, 5), mas suas coordenadas 3D podem ser (0, 0, 0) e (0, 0, 5).

5.1.4 Coordenadas Cartesianas

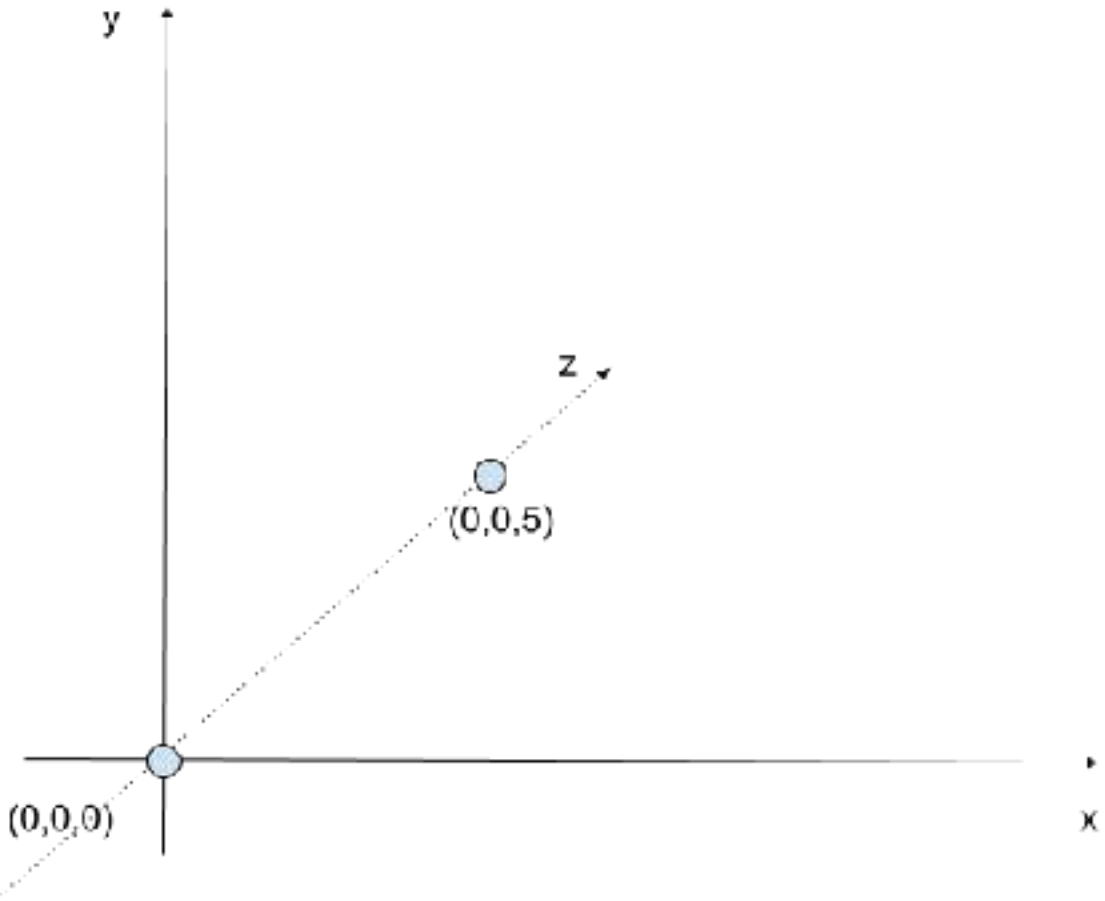
Uma das maneiras mais simples de entender a relação entre coordenadas é usar o sistema de coordenadas cartesianas. Por exemplo, podemos desenhar o primeiro exemplo de Wesnoth assim:



A vantagem de visualizar coordenadas como esta é que podemos usar operações geométricas comuns. Suponha que queremos obter a distância entre essas duas unidades. Ao criar um triângulo retângulo a partir dessas duas unidades, podemos usar o teorema do pitagórico para calcular a inclinação do triângulo. Devido à maneira como criamos este triângulo, esse oblíquo representará a distância entre essas duas unidades:



As coordenadas 3D também podem ser desenhadas adicionando outro eixo. O exemplo de jogo 3D que temos acima pode ser o seguinte:



Observe que, embora cada jogador tenha as mesmas coordenadas X e Y, elas existem em posições diferentes no gráfico.

5.1.5 Porto de Visão

O monitor exibe uma imagem 2D na tela do tablet. Portanto, é **impossível para um monitor renderizar cenas 3D diretamente. Em vez disso, o mundo 3D deve ser convertido em uma cena 2D, assim como tirar fotos. Os jogos geralmente têm essa função e são frequentemente chamados de alguma variação do World ToScreen.** Às **vezes, quando se trata de técnicas de programação, como exibir texto acima da cabeça do jogador, você precisa escrever esse código sozinho. No Capítulo 5.9, discutiremos como escrever esse código para qualquer jogo.**

Um aspecto fundamental da transição de 3D para 2D é que o jogo escolherá uma porta de visão ou uma visão do mundo do jogo. Geralmente é a visão do jogador atual , mas em jogos que suportam câmeras gratuitas, isso pode estar em qualquer lugar. Para esse ponto de vista, o jogo calculará a profundidade de todos os objetos na cena. Ele também desenha objetos mais distantes "atrás" dos objetos mais próximos. Por exemplo, na vista abaixo, o jogo primeiro desenha o edifício no fundo. Ele então determina que as árvores estão localizadas "na frente" do edifício no ponto de vista atual e as desenha no topo do edifício. Dessa forma, o jogo cria uma ilusão profunda.

5.1.6 Esporte 3D

O custo de computação operacional no espaço 3D é muito alto. Por isso, os jogos geralmente pegam vários atalhos para otimizar o desempenho. Um desses atalhos sempre coloca o jogador no ponto de origem ou ponto (0,0, 0). Dessa forma, todas as distâncias e ângulos do objeto podem ser calculados recuperando apenas suas coordenadas, em vez de ter que subtrair a posição do objeto da posição do jogador. Entretanto, se o jogador for

Se estiverem presos em (0,0,0), eles não serão capazes de se mover. Para criar a ilusão de movimento, alguns jogos fazem o mundo inteiro girar em torno do jogador. Por exemplo, se você pressionar o botão para frente, o jogo reagirá movendo o mundo inteiro em sua direção, em vez de deixar seu jogador se mover para a frente. Nem todos os jogos funcionam assim, mas vários jogos conhecidos usam esse modo.

5.2 Wallhack (memória)

5.2.1 Metas

Agora que estamos mudando para uma nova dimensão, precisamos mudar para um novo objetivo. Os próximos capítulos abordarão o terror urbano 4.3.4. Este jogo é um FPS baseado no motor Thor Hammer.

Assim como o Wesnoth, este jogo é de código aberto e não integra antitrapaça. Ele também funciona bem em hardware de baixa especificação. Ao contrário da Wesnoth, a embalagem de chocolate é ruim . Portanto, a melhor maneira de instalar o jogo é baixar e executar o programa de instalação do site.

Você precisará ativar a aceleração 3D no VirtualBox para fazer o jogo funcionar. Dependendo do hardware do seu computador, seu computador pode não ser capaz de executar jogos 3D em uma máquina virtual. Nesse caso, você tem várias opções. Alguns são melhores que os outros:

1. Explore outro programa de gerenciamento de máquinas virtuais, como VMWare ou Hyper-V.
2. Use outra máquina como computador dedicado ao hacker e isole-a da rede doméstica.
3. Encontre outro jogo de metas com menos requisitos e siga os conceitos dos capítulos seguintes.
4. Participe seu disco rígido e inicialização dupla. Mesmo que você criptografe sua unidade pessoal, ferramentas maliciosas podem acessar seus dados pessoais.
5. Executar alvos e ferramentas em sua máquina pessoal, na esperança de que nenhuma intenção maliciosa aconteça.

5.2.2 Identificação

Nosso objetivo neste capítulo é criar um wallhack, um hacker que nos permite ver outros jogadores através das paredes. Não vamos modificar

jogo. Em vez disso, usaremos a lógica de renderização do jogo e modificaremos partes da memória do jogo.

5.2.3 Compreensão

Em jogos 3D, testes de profundidade são usados para determinar quando um projeto deve ser visível no campo de visão do jogador. Por exemplo, se o jogador estiver atrás da parede , o teste profundo dirá a lógica de renderização do jogo para não desenhar o jogador.

Todos os wallhacks são executados com base no princípio de desativar testes profundos. Uma maneira é ganhar a biblioteca gráfica do jogo e desativar os testes de profundidade por meio da função da biblioteca. Descreveremos essa abordagem no próximo capítulo. Neste capítulo , contaremos com a lógica de renderização incorporada ao jogo para atingir nossos objetivos.

O jogo exige o desenho de muitos objetos dinâmicos, incluindo jogadores, armas e ativos de mapa, como portas. Esses objetos são frequentemente chamados de entidades. Para simplificar o desenvolvimento e melhorar o desempenho, os jogos geralmente usam a mesma função para desenhar todas essas entidades.

Entretanto, essas entidades geralmente têm considerações de renderização diferentes. Um jogo pode querer desenhar sombras em personagens, mas não em entidades estáticas, como portas que podem ser abertas. Os jogos geralmente têm uma estrutura para cada entidade, e essas renderizações são armazenadas na estrutura da entidade. Quando uma entidade é renderizada, o jogo verifica o membro e renderiza a entidade com base nela.

Para algumas entidades, como poças d'água ou vidro, o jogo pode desativar os testes profundos. Portanto, os membros de renderização na classe física terão valores de teste de profundidade desativados. Se pudermos encontrar a função responsável por desenhar entidades e modificar todas as entidades para incluir esse valor de teste de profundidade desativado, o jogador aparecerá através da parede.

5.2.4 Definição de metas

Todos os jogos baseados no motor Thor Hammer têm um console. No jogo, você pode acessar este console clicando no botão de sinal de onda (~). Esta console permite que você execute comandos, como mover o player ou alterar o mapa. Esses comandos geralmente começam com uma barra reversa (\) e podem ser concluídos automaticamente pressionando a tecla tab. Alguns comandos úteis para o nosso propósito são:

- **\devmap abbey - Iniciando um mosteiro de mapas que habilita trapaças**

- *\g_gametype 0–Defina o modo de jogo padrão para uma corrida de morte*
- *\bot_enable 1–Ative robôs para participar do jogo*
- *\Reload–Reiniciar o mapeamento atual*
- *\addbot boa 1–Adicionar robôs*

Além desses comandos, também podemos alternar facilmente o jogo para o modo janela pressionando Alt+Enter.

5.2.5 Elementos de mapa de posicionamento

Navegando pelos comandos disponíveis, podemos encontrar vários comandos de desenho em\r_:

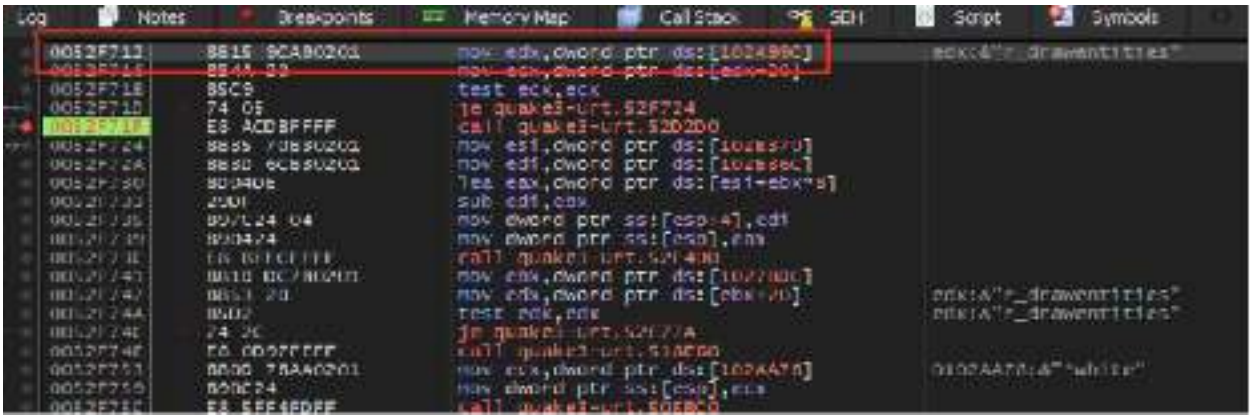
O comando mais importante para nós é r_drawentities. Ao definir esse valor para 0, nenhuma entidade será desenhada no jogo, incluindo nossos jogadores:

Podemos assumir que o código do jogo é o seguinte:

```
Se (R_Draw_Entidades==1){
Draw_Entidades();
}
```

Para encontrar esse código, usaremos o mecanismo de fraude para encontrar o endereço da variável que salva o valor R_Drawentites. Podemos alternar o valor de r_drawentities de 0 para 1 no console para restringir a faixa desse valor. Então podemos usar pontos de interrupção ao acessar

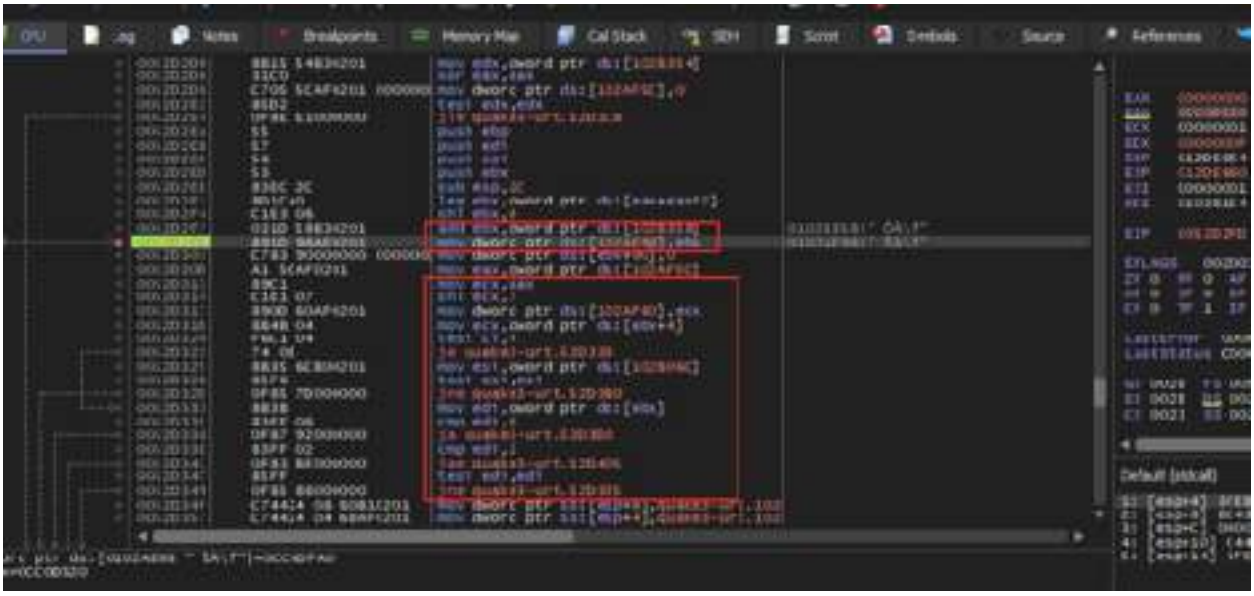
x64dbg é usado para localizar o código que acessa esse valor. Pontos de interrupção devem aparecer nos seguintes códigos:



Podemos ver que o valor do r_drawentities é carregado no ecx e depois testado. O próprio registro de teste compara o valor do registro com 0. Se esse valor for igual a 0, o jogo pula a chamada 0x52F717. É provável que essa chamada seja responsável por desenhar entidades no jogo. Podemos confirmar isso cancelando esta ligação. Quando estiver nop' d, o jogo não desenhará nenhuma entidade.

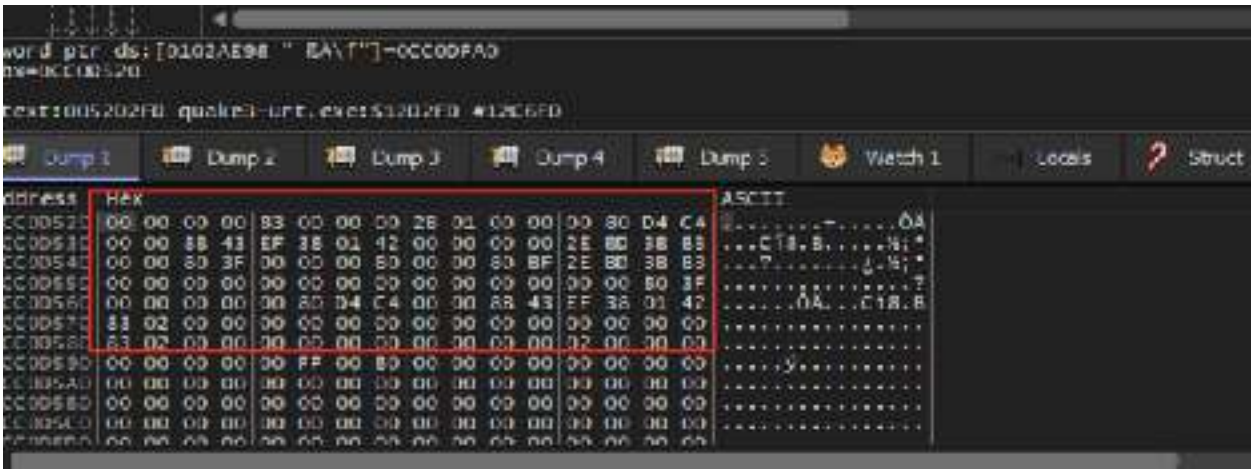
5.2.6 Entidades e Renderização

Se entrarmos na chamada em 0x52F717, veremos o seguinte código:



Podemos ver no segundo bloco destacado que os valores são carregados em vários registros e comparados com certos valores. Se esses valores forem iguais, o jogo salta para diferentes posições e executa códigos de renderização diferentes. Se observarmos atentamente, podemos ver

O registro é baseado no valor do endereço armazenado em ebx. Se olharmos para cima para o primeiro bloco destacado, encontraremos o local mais próximo para configurar o ebx. Agora sabemos que no endereço 0x52D2FD, ebx contém a entidade atual que tem maior probabilidade de ser apresentada. Se definirmos um ponto de interrupção neste endereço e observarmos o endereço ebx no dump, veremos um bloco de dados:



Como esse bloco de dados é isolado de outros dados e está em uma seção contínua , podemos assumir que ele representa um certo tipo de estrutura. Por exemplo, pode parecer:

```
Entidade estrutural{
Tipo int;
INTRENDER_tipo; Posição
do ponto flutuante[3];
...}
```

Para determinar onde uma estrutura salva o tipo de renderização, precisamos revertê-la.

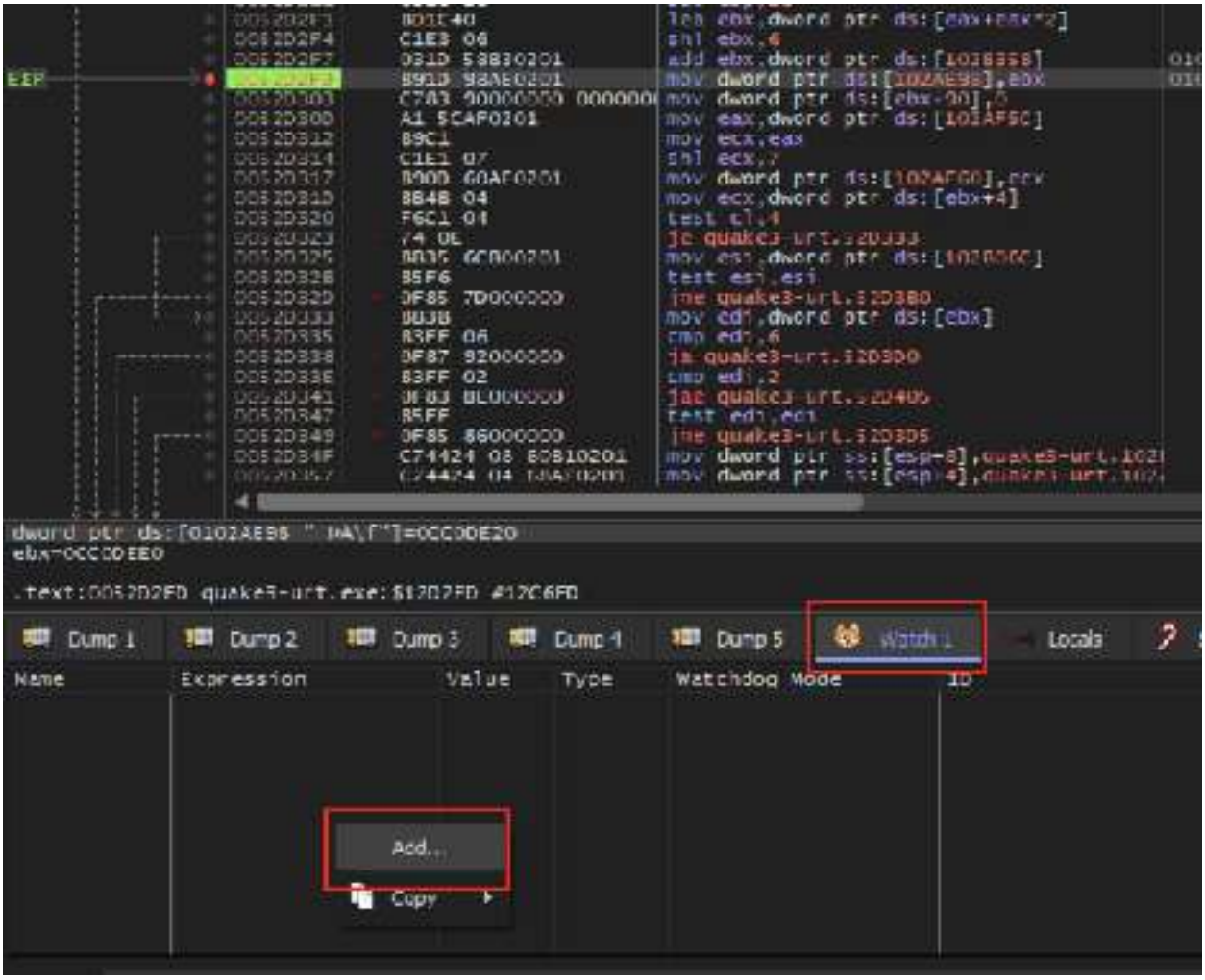
5.2.7 Estrutura de Entidade Reversa

Há muitas maneiras de reverter uma estrutura desconhecida no jogo. Uma abordagem é criar um conjunto de dados de valores válidos e então fazer inferências com base nesses valores. Por exemplo, se todas as estruturas contêm um membro crescente, podemos presumir que esse membro é usado como algum tipo de contador.

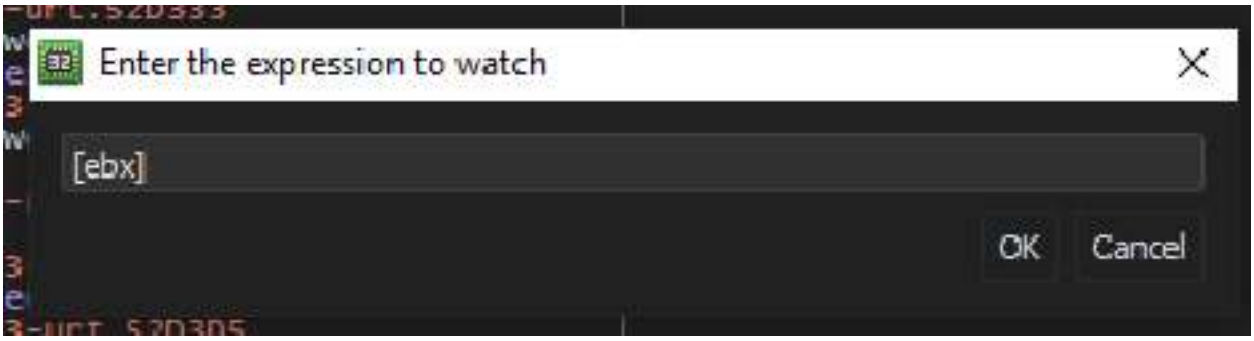
Neste caso, nosso objetivo não é reverter completamente a estrutura da entidade, mas apenas revertê-la a um nível suficiente para encontrar variáveis do tipo render. Como encontramos o código responsável por desenhar a entidade, podemos definir um ponto de interrupção nesse código e observar a estrutura da entidade. Como discutimos na seção anterior, no endereço 0x52D2FD, o ebx salva o endereço da entidade atual a ser renderizada.

Você notará que cada vez que um ponto de interrupção é atingido, o ebx contém um valor diferente . Embora possamos rastrear manualmente o ebx no dump sempre que encontrarmos um ponto de interrupção, uma abordagem mais conveniente é usar o recurso Watch do x64dbg. Adicionar uma expressão ao painel de observação nos permite observá-la independentemente do dumping. Neste caso, podemos visualizar a expressão [ebx] e sempre visualizar o valor atual do endereço em ebx.

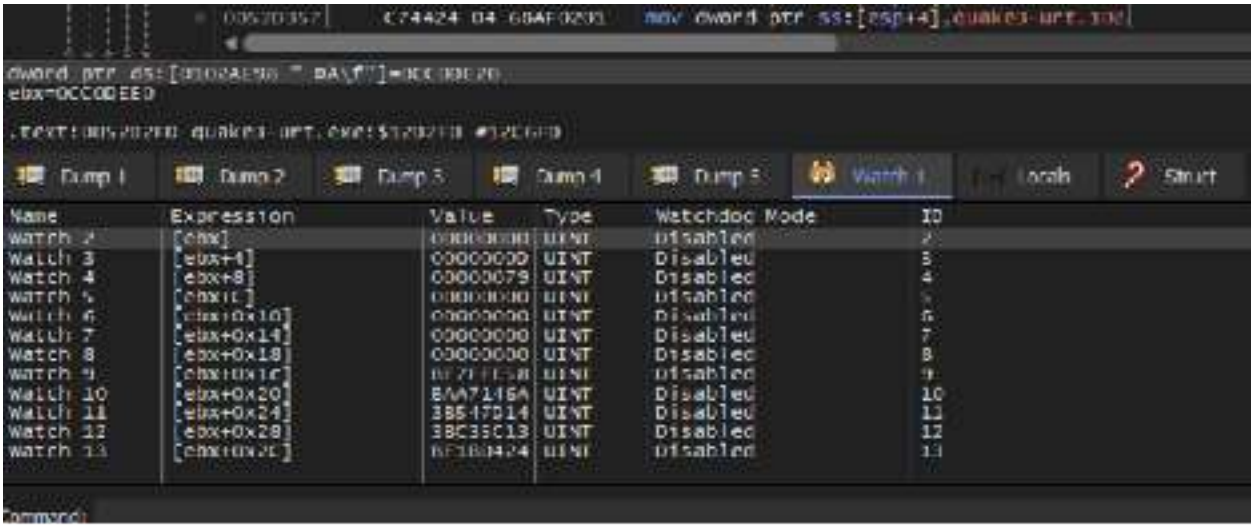
Para adicionar o valor que deseja monitorar, abra o painel de monitoramento (perto do dump), clique com o botão direito do mouse e selecione Adicionar:



No modo que aparece, escreva a expressão. Neste caso, queremos começar com [ebx]:



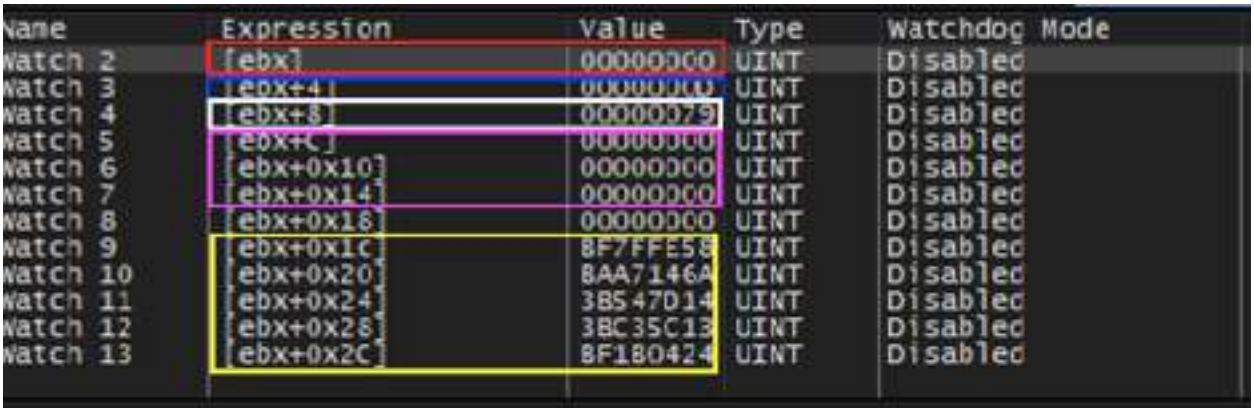
Também queremos observar o primeiro bloco da estrutura física. Agora, assumiremos que todos esses valores tenham 4 bytes de comprimento. Adicione um relógio a [ebx+4] para [ebx+2C]. Após a conclusão, o painel do relógio deve ser o seguinte:

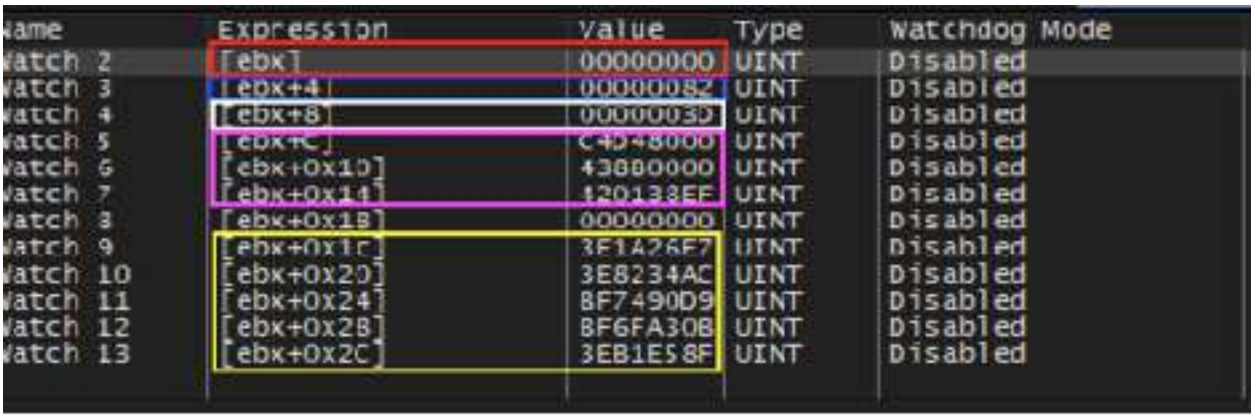


Depois de concluir todas essas configurações, desative os pontos de interrupção e carregue-os no mapa com água. O Mosteiro do Mapa tem uma fonte perto do canto superior esquerdo do mapa. Certifique-se de que você esteja de frente para a água e veja o chão abaixo.

Após concluir todas essas configurações, reativar o ponto de interrupção em 0x52D2FD e ele deverá aparecer imediatamente. Após observar os valores do painel de observação, continue . Depois de observar várias iterações, você deve começar a notar algumas tendências.

ame	Expression	Value	Type	Watchdog Mode
atch 2	[ebx]	00000000	UINT	Disabled
atch 3	[ebx+4]	00000000	UINT	Disabled
atch 4	[ebx+8]	00000078	UINT	Disabled
atch 5	[ebx+C]	00000000	UINT	Disabled
atch 6	[ebx+0x10]	00000000	UINT	Disabled
atch 7	[ebx+0x14]	00000000	UINT	Disabled
atch 8	[ebx+0x18]	00000000	UINT	Disabled
atch 9	[ebx+0x1C]	BF7FFDCA	UINT	Disabled
atch 10	[ebx+0x20]	BB25B2C4	UINT	Disabled
atch 11	[ebx+0x24]	3B9C493A	UINT	Disabled
atch 12	[ebx+0x28]	3BE0A705	UINT	Disabled
atch 13	[ebx+0x2C]	BF1AFADF	UINT	Disabled





O valor [ebx](vermelho) é sempre exibido como 0. Os valores [ebx+4](azul) parecem alternar entre 0xD, 0x40, 0x82 e 0x83. O valor [ebx+8](branco) parece continuar aumentando, de 0x79 para 0x80 para 0x81, e assim por diante. Os valores destacados em rosa parecem alternar entre valores aparentemente aleatórios e 0. Da mesma forma, os valores destacados em amarelo parecem aleatórios, mas estão sempre associados a [ebx+8].

Todos esses dados podem ser confusos, mas podemos entendê-los eliminando valores com os quais não nos importamos. Sabemos que há pelo menos três entidades na tela: nosso modelo de jogador, nossas armas e água. Podemos presumir que pode haver outras entidades, como portas. Como a maioria dessas entidades tem muitas semelhanças , queremos encontrar dados relativamente consistentes entre pelo menos duas entidades. No entanto, também sabemos que algumas entidades não devem compartilhar esse valor.

Com esse modelo, podemos eliminar [ebx](vermelho) porque é sempre 0. Também podemos eliminar [ebx+8] (branco) porque ele é único para cada entidade. Os valores rosa e amarelo são únicos para cada objeto. Isso nos deixa [ebx+4] (azul), que alterna entre 0xD, 0x40, 0x82 e 0x83. Agora, adivinharemos que esse é o nosso valor de renderização e investigaremos cada valor.

5.2.8 Modificar o valor da renderização

Se definirmos um valor de [ebx+4] para cada entidade, ela será sobrecarregada na próxima vez que chamarmos a função "Entidade de Desenho". A entidade parece estar sendo carregada no ebx de outro local. Portanto, a maneira mais simples de explorar o valor de renderização hipotético é ganhar a posição 0x52D2FD e definir [ebx+4] para cada entidade. Podemos criar esta caverna de código em x64 dbg, mas para facilitar o teste de vários valores, criaremos ganchos em DLL.

Usaremos a mesma estrutura de gancho discutida no Capítulo 3.4. Nosso gancho estará localizado em 0x52D2FD porque sabemos que o ebx conterá o valor correto naquele ponto. Nosso gancho em si é relativamente simples: salvaremos o registro, definimos o valor [ebx+4], restauramos o registro e executamos a instrução mov original:

```
DWORD Ret_Adress=0x0052D303;

__declspec(naked)void codecave(){_asm
{
Pushad
mov dword ptr ds: [ebx+4],??? Rebote

mov dword ptr ds: [0x102aa98], ebx jmp
Ret_Adress

}}
```

Para o nosso primeiro valor, vamos começar pelo topo e tentar 0x83:

```
mov dword ptr ds: [ebx+4], 0x83
```

Depois que o DLL for injetado, você retorna ao jogo e notará que nada mudou. Da mesma forma, se você experimentar o 0x40, poderá notar que algumas sombras parecem diferentes, mas tudo parece muito parecido. Em seguida, vamos tentar 0xD:

```
mov dword ptr ds: [ebx+4], 0xD
```

Você deve notar imediatamente que o modelo do personagem agora está exibido como transparente na frente da câmera:

Este é um bom sinal de que os testes de profundidade podem ter sido desativados. Em seguida, mude para o modo terceira pessoa (cg_thirdperson) e adicione alguns robôs. Ao caminhar, você deve notar que agora pode ver todos os robôs através das paredes:

Dessa forma, definimos com sucesso os valores de renderização de todas as entidades para desativar os testes de profundidade. Podemos ver outras entidades, como armas e escadas, aparecendo através das paredes.

Uma melhoria é reativar os testes profundos do nosso modelo de jogador para que o modo de primeira pessoa não seja interrompido. Para isso, você precisa determinar a estrutura do jogador e seu jogador atual.

O código-fonte completo deste hacker pode ser encontrado no Apêndice A para comparação.

5.3 Wallhack (OpenGL)

5.3.1 Objetivos

Nosso capítulo tem como objetivo o terror urbano 4.3.4.

5.3.2 Visão Geral

A maioria dos jogos é renderizada usando bibliotecas gráficas externas. As duas bibliotecas gráficas mais populares são DirectX e OpenGL. Ambas as bibliotecas são carregadas dinamicamente pelo jogo. Uma vez carregados, o jogo chama funções nessas bibliotecas. Por exemplo, usando OpenGL, o jogo pode usar a função glDrawElements para desenhar uma série de elementos a partir de dados armazenados em um array. Como essas bibliotecas são externas, os desenvolvedores de jogos não precisam implementar a lógica de renderização por conta própria.

5.3.3 Identificação

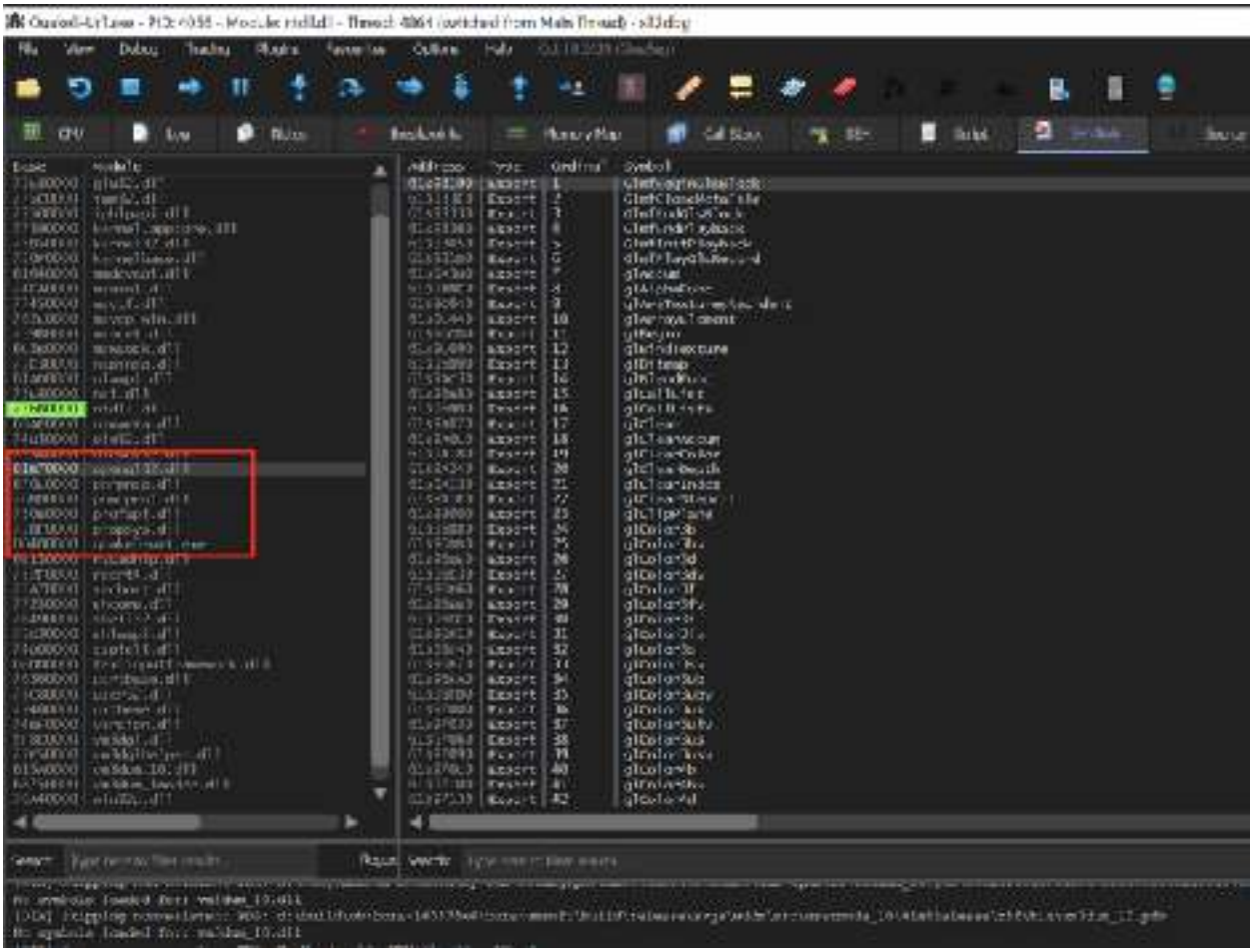
Nosso objetivo neste capítulo é criar um wallhack ganhando a biblioteca gráfica do jogo e modificando sua lógica para exibir entidades através da parede.

5.3.4 Compreender

Tanto o DirectX quanto o OpenGL têm funções de renderização diferentes e exigem métodos de gancho diferentes. Nosso primeiro objetivo é identificar as bibliotecas que o jogo está usando. Como cada biblioteca tem várias funções para lidar com renderização e coloração, precisamos encontrar uma função para renderizar o jogo. Depois de identificar a função, podemos enganá-la e desativar o teste de profundidade usando a caverna de código. Isso resultará na renderização de todas as entidades, independentemente de sua localização no mundo 3D.

5.3.5 Galeria de Posicionamento

Como as bibliotecas gráficas são carregadas dinamicamente, suas funções devem ser exibidas ao executável principal. A maioria dos depuradores permite que você visualize todas as bibliotecas carregadas em arquivos executáveis quando adicionado. No x64dbg, essas informações são coletadas na aba de símbolos.



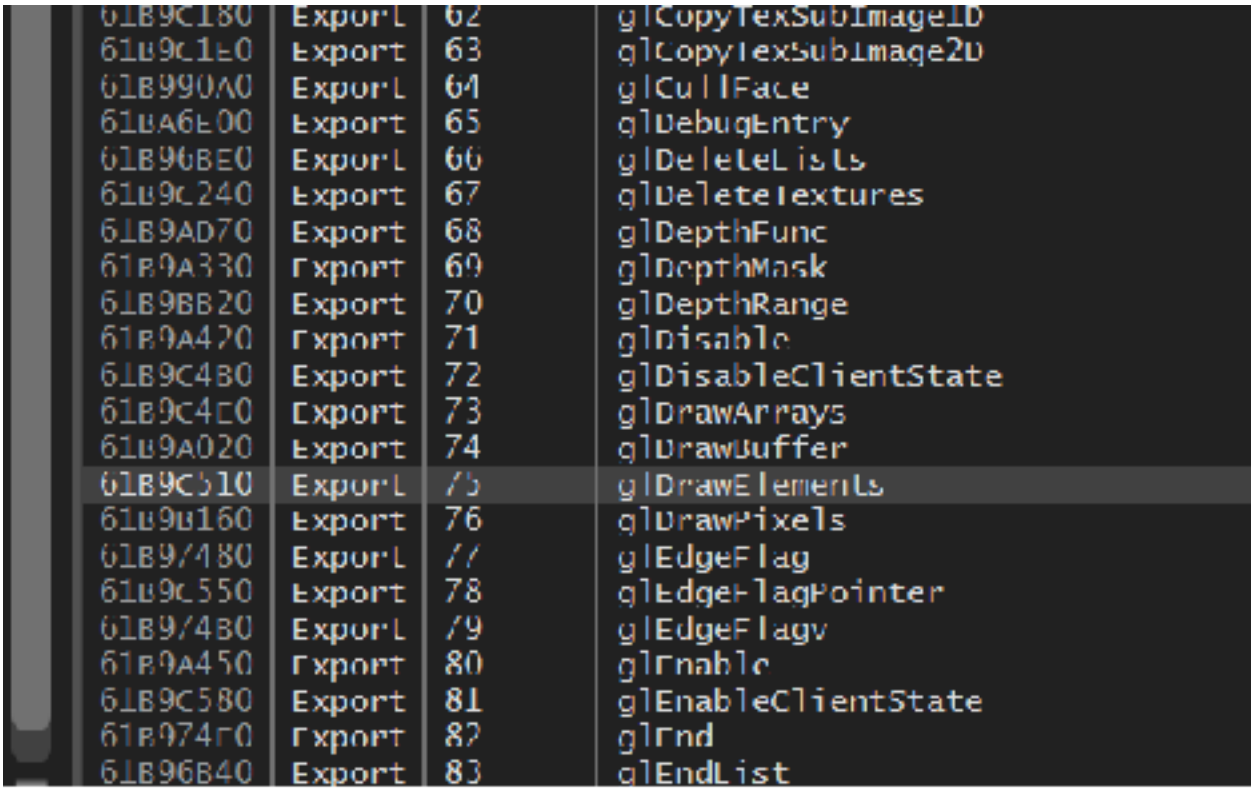
Como podemos ver nos elementos destacados, o opengl32.dll está sendo carregado no processo do jogo. Ao selecionar o módulo OpenGL, podemos ver que ele exporta muitas funções relacionadas a desenhos. A partir dessas informações, podemos concluir que este jogo está usando OpenGL para renderizar seus gráficos.

5.3.6 Função de localização e desenho

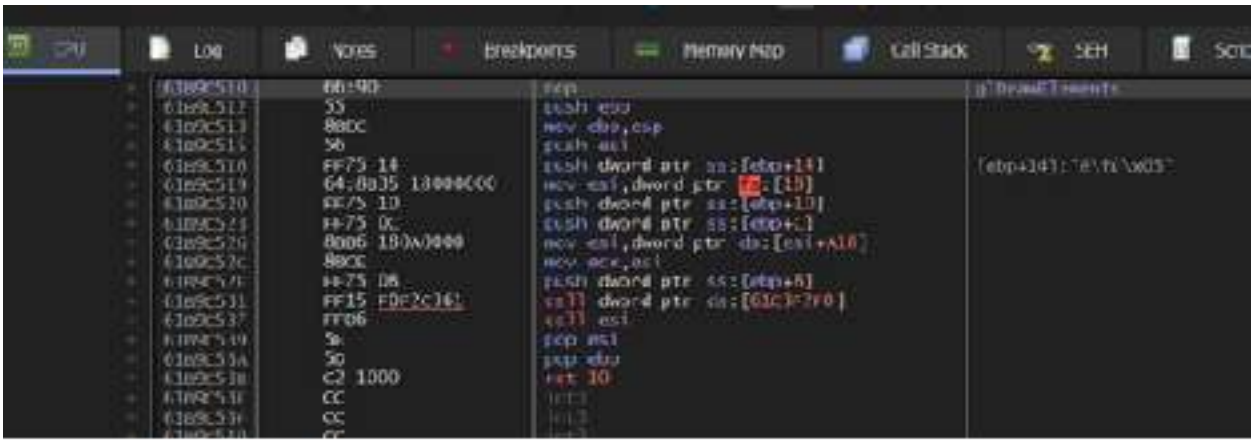
O OpenGL tem vários métodos de renderização, e diferentes jogos usam métodos diferentes. Por exemplo, jogos mais antigos podem usar glBegin, glVertex e glEnd; Alguns jogos podem usar GLDRAWRAYS Outros podem usar glDrawElements. Alguns até usam uma combinação desses métodos para renderizar aspectos diferentes, como glDrawElements para modelos de jogadores e glBegin para efeitos de tela como sangue.

Geralmente, glBegin, glVertex e glEnd não são usados em jogos modernos porque essas funções são consideradas não recomendadas. Portanto, não estamos nos concentrando nesses recursos agora. Em vez disso, primeiro estudaremos glDrawElements, porque é uma função comumente usada. Devido à maneira como o OpenGL funciona, se o jogo usar essa função, esperamos que ela seja chamada continuamente.

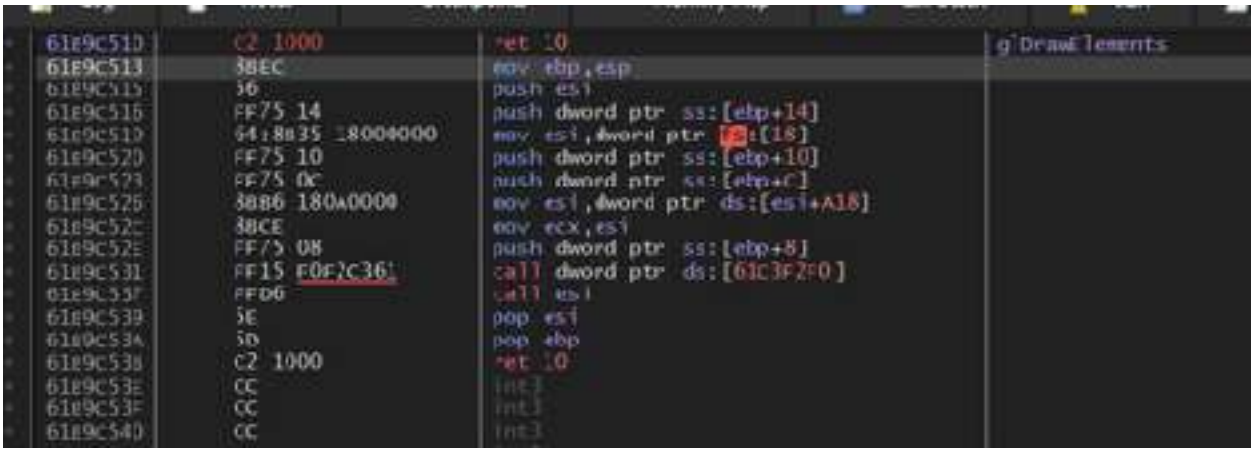
Ao rolar para baixo para a Exportação glDrawElements na aba Symbols, podemos ver que o OpenGL o exporta para o processo, embora isso não garanta que ele esteja em uso:



Clique duas vezes na entrada de exportação e x64dbg exibirá a função:



Em seguida, podemos iniciar um jogo e definir um ponto de interrupção no glDrawElements. Você notará que ele pop-up imediatamente e continuará sempre que o jogo for retomado. Isso mostra bem que a função é responsável por apresentar entidades. Para verificar se isso é verdade, podemos substituir a primeira instrução pela declaração ret que vemos no final da função. O efeito disso é que o código de chamada é devolvido imediatamente sem executar nenhuma lógica glDrawElements:

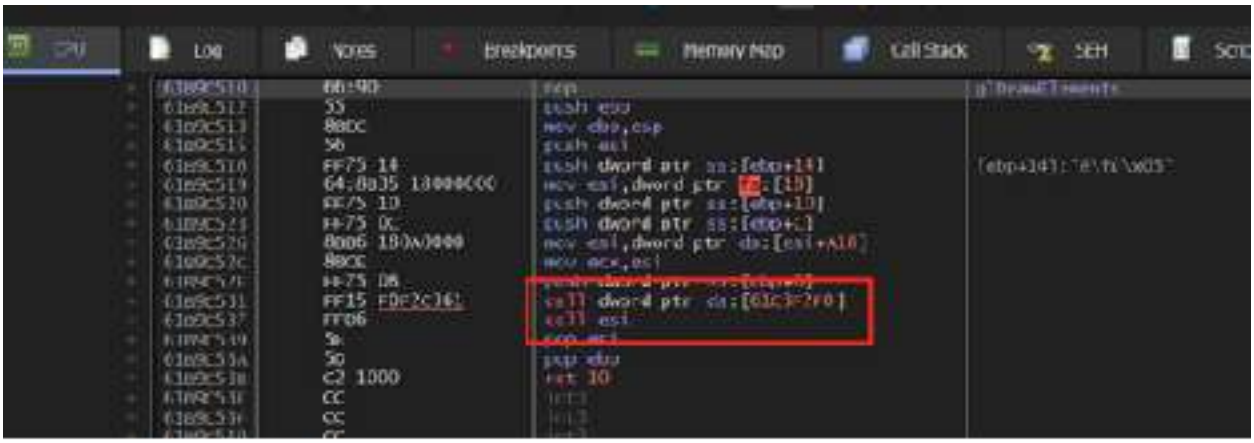


Se você retomar a execução posteriormente e tentar jogar, notará que nenhuma nova entidade é apresentada na tela:

Isso nos dá fortes evidências de que o terror urbano está usando glDrawElements para mostrar entidades.

5.3.7 Elemento GLDRAWE gancho

Ao verificar a função glDrawElements, podemos ver que ela tem poucas instruções. Dada a complexidade de renderizar entidades na tela, a maior parte do código deve ser incluída em duas chamadas próximas ao final da função:



Portanto, se fixarmos uma instrução antes dessas chamadas, deveríamos ser capazes de atingir o objetivo de desativar os testes profundos. Uma boa instrução candidata é o mov em 0x61B9C526.

Como o OpenGL é carregado dinamicamente, nosso método de gancho deve ser um pouco diferente. Primeiro, precisamos garantir que o OpenGL esteja realmente carregado. Isso não acontece porque injetamos DLL no aplicativo quando ele é iniciado pela primeira vez. Depois de garantir que o OpenGL esteja carregado, precisamos descobrir onde ele está carregado. Depois de determinar o local do OpenGL, podemos determinar a localização do GLDRAWENTIES no módulo OpenGL.

Usaremos a combinação de tecnologias que exploramos em capítulos anteriores. Para resolver o problema em que o OpenGL não é carregado quando nosso DLL é injetado, criaremos um thread para lidar com a lógica de gancho. Isso nos permitirá criar um loop infinito esperando que o OpenGL carregue, semelhante ao thread que vemos no Capítulo 3.3:

```
Se(fdwReason==DL_Process_Attach){
Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL, 0, NULL);

}
```

Em nossos threads, criaremos um loop infinito de chamar GetModulehandel. Esta API retorna o cabo ou endereço do módulo que carrega o módulo. Se o módulo não estiver carregado, ele retornará a NULL:

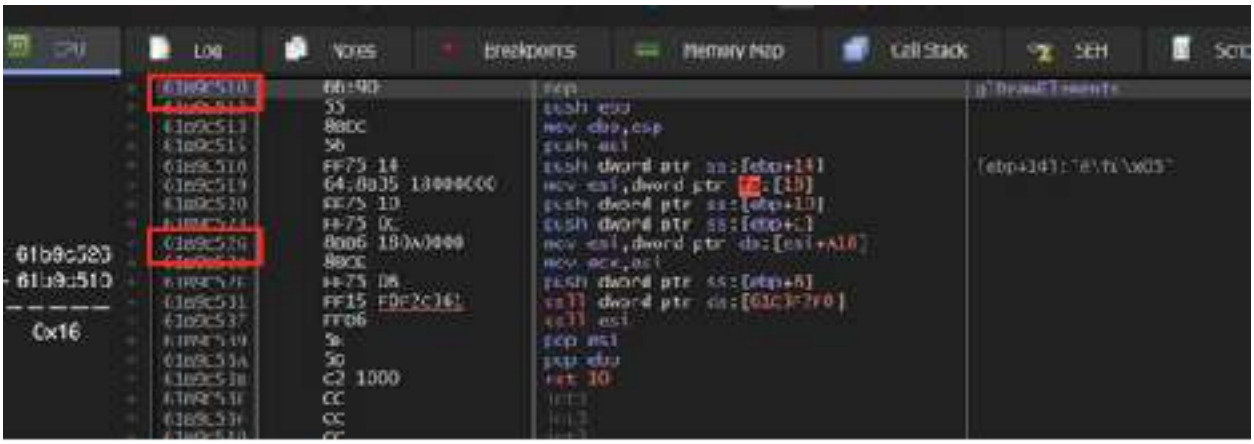
```
HMODULE Openglhandele=NULL;

void Injectid_THEAD(){while(
true){
```

```
Se(Openglhandel==NULL){
Opengll e=GetModuleHandle (L"opengl32.dll");}

... Sono (1);
```

Quando temos o local do OpenGL, podemos encontrar a localização do glDrawElements. Para isso, usaremos a API GetProcAddress. Quando um módulo e o nome de uma função são dados, a API retorna o endereço da função:

```
unsigned char*hook_localização;
...
if(Openglhandel!=NULL){
Hook_Localização=(unsigned char*) GetProcadernets (Openglândle, "
glDrawElements");
```

Esta API retornará a localização da primeira instrução na função, neste caso nop. Como queremos ganhar o comando mov, podemos subtrair a distância dela da primeira instrução e adicionar a diferença ao resultado do GetProAddress. A distância entre essas duas instruções é sempre a mesma porque fazem parte do código de base e não são carregadas dinamicamente.



Este deslocamento pode ser adicionado diretamente à variável hook_localização para obter nossa posição:

```
Hook_Localização+=0x16;
```

Por fim, podemos suspender o código, como fizemos nos capítulos anteriores:

```
Projeto Virtuo* (void*)Hook_Locasione, 5. página _Express_Readwrite,
&old_protect);
*Hook_Localização=0xE9;
*(DWORD*)(hook_localização+1)=(DWORD)&codecavidade-((DWORD)hook_localização+5
); *(hook_localização+5)=0x90;
```

5.3.8 Ponteiro de função

Depois de conectar o glDrawElements, podemos começar a processar o código caverna. Nosso objetivo é desativar testes de profundidade ao desenhar elementos. Para isso, podemos usar uma função OpenGL chamada glDepthFunc. glDepthFunc permite que você configure a função para comparação de profundidade quando a OpenGL tenta renderizar a tela. Esses valores podem ser variados, mas estamos interessados em GL_LEQUAL (desenhar se o elemento estiver antes de outro elemento) e GL_ALWAYS (sempre desenhar).

Para nosso Wallhawk, definimos a função de profundidade para GL_ALWAYS antes de desenhar qualquer elemento. Isso terá o efeito de fazer com que todos os elementos apareçam sempre, independentemente de sua posição real no espaço 3D.

Primeiro, precisamos encontrar o glDepthFunc. Podemos usar o GetProCADRESS de uma forma semelhante ao glDrawElements. Entretanto, nosso objetivo ao chamar GetProcAddress não é encontrar um endereço para ganhar, mas armazenar o endereço da função de uma forma que podemos chamá-lo em uma caverna de código. O método mais simples é através do ponteiro de função.

Assim como o ponteiro que usamos nos capítulos anteriores, o ponteiro de função aponta para um endereço. Entretanto, diferentemente dos ponteiros que sempre usamos para modificar dados e código, também podemos declarar um ponteiro apontando para uma função. Então podemos chamar essa função ou endereço, assim como chamamos qualquer outra função C++.

Para declarar ponteiros de função, precisamos conhecer a definição da função original . A definição de uma função inclui seu tipo de retorno e parâmetros. Podemos obter estas informações no site do Khronos Group:

```
void GLDEPTHUNC (GLenum func);
```

Verificando o arquivo de cabeçalho gl.h, podemos descobrir o que é GLenum:

```
typedef int GLenum sem símbolo;
```

Até agora, podemos definir a função glDepthFunc da seguinte forma:

```
void glDepthFunc (int sem sinal) = NULL;
```

Em seguida, modificaremos esta declaração para que ela sirva como um ponteiro para esta função:

```
void(*glDepthFunc) (int sem sinal)=NULL;
```

Agora podemos atribuir isso aos resultados da GETPROCDRESS:

```
glDepthFunc = glDepthFunc (OpenGLDEPTHFUNC, "glDepthFunc");
```

No entanto, se tentarmos construí-lo, teremos os seguintes erros:

```
Erro C2440: '=': Não pode ser convertido de 'FARPROC' para 'void(__CDCL*)(unsigned
int)'
Mensagem: Esta conversão requer conversão de estilo Reinterpres_cast, conversão de
estilo C ou conversão de estilo funcional
```

Como vimos nos capítulos anteriores, precisamos converter corretamente os resultados do GetProcAddress para que o compilador possa entender como convertê-los. Podemos usar mensagens de erro para determinar rapidamente como converter resultados:

```
GLDEPTHUNC=(void(__CDCL*)(unsigned int)) (OpenGLDEPTHUNC, "GLDEPTHUNC");
```

5.3.9 Caverna de Código glDrawElements

Nossa caverna de código será semelhante à caverna de código que escrevemos antes. Começaremos com nosso esqueleto e restauraremos o código original:

```
DWORD Ret_Adress=0;

__declspec(naked)void codecave(){_
asm{
Empurrar}

... _asm{
```

```
Rebote
mov esi, dword ptr ds: [esi+0xA18]jmp Ret_Adress

}
```

Ao contrário dos capítulos anteriores, não temos uma posição estática para pular. Em vez disso, precisamos calcular a posição de retorno da mesma forma que calculamos a posição do gancho. Em nossos threads, depois que atribuímos a posição do gancho, também podemos atribuir dinamicamente a posição de retorno:

```
Ret_Adress=(DWORD) (hook_localização+0x6);
```

Com nossa estrutura, agora podemos adicionar chamadas ao GLDePTHUNC. Primeiro, precisamos encontrar o valor de GL_ALWAYS. Podemos encontrá-lo no arquivo de cabeça gl.h:

```
#Definição GL_ALWAYS 0x0207
```

Em seguida, podemos chamar glDepthFunc para desativar o teste profundo. Como é um ponteiro de função, precisamos decifrar o ponteiro de referência para chamar a função:

```
(*glDepthFunc) (0x207);
```

Nosso código é o seguinte:

```
#include<Windows.h>

HMODULE Openglhandele=NULL;

void(*glDepthFunc) (int sem sinal)=NULL;
unsigned char*hook_localização;


DWORD Ret_Adress=0;
DWORD old_protect;

__declspec(naked)void codecave(){_
asm{
Empurrar}
```

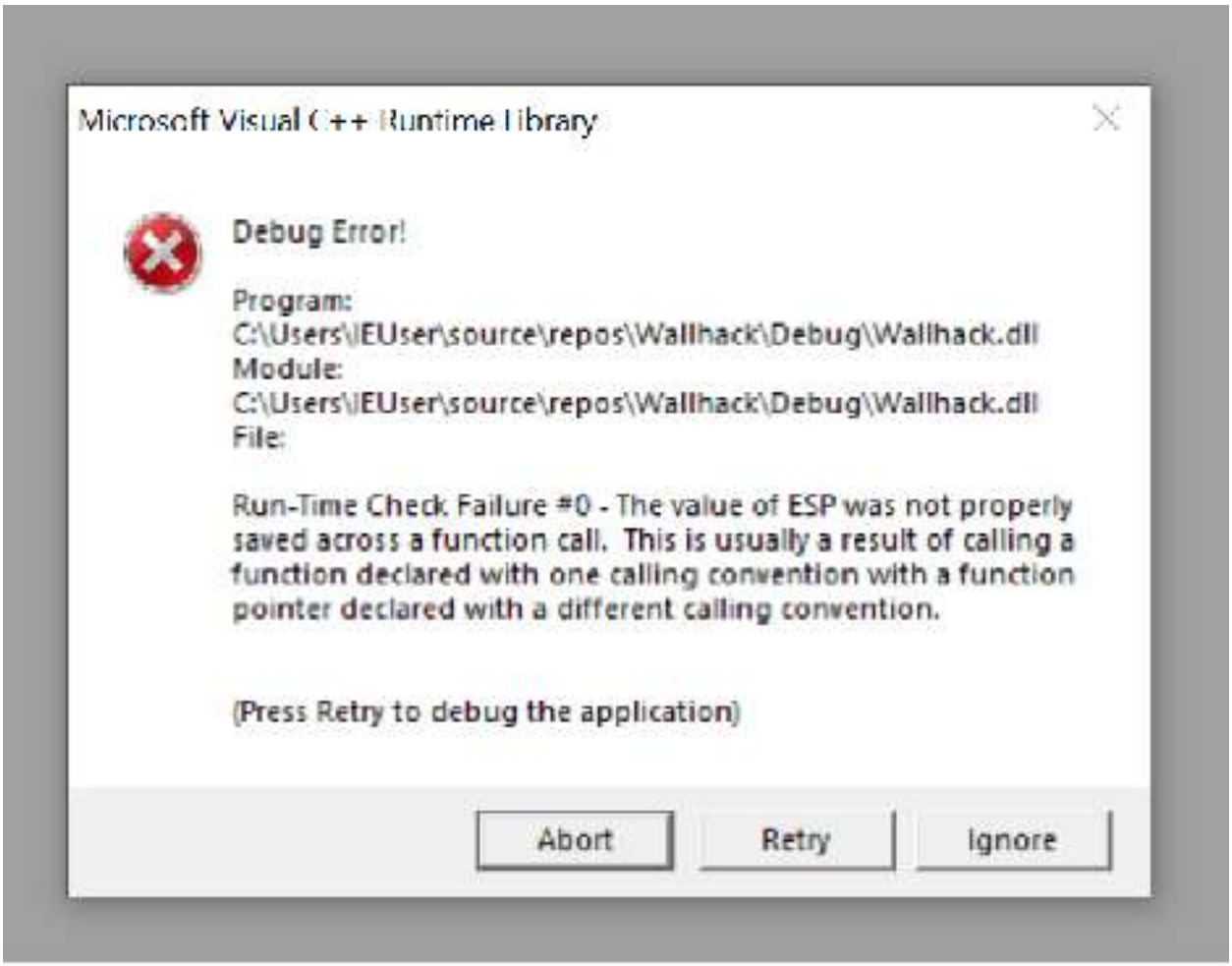
```
(*glDepthFunc)(0x207);

_asm{
Rebote
mov esi, dword ptr ds: [esi+0xA18]
jmp Ret_Adress
}
}

void injectid_THREAD(){
while(true){
Se(Openglhandel==NULL){
Opengll e=GetModuleHandle (L"opengl32.dll");
}

if(Openglhandel!=NULL){
glDepthFunc=(void(__cdecl*) (sem sinal
int) GetProcadodres (openGLHandle, "glDepthFunc");

Hook_Localização=(unsigned char*)GetProcadodres (OpenGLADLE,
"GLDRAWEMENTS");
Hook_Localização+=0x16;

Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_Excessionismo, &ol_
Projeto);
*Hook_Localização=0xE9;
*(DWORD*)(hook_localização+1)=(DWORD)&codeave-
((DWORD)hook_localização+5);
*(hook_localização+5)=0x90;

Ret_Adress=(DWORD) (hook_localização+0x6);
}

Sono (1);
}
}

BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

Se(fdwReason==DL_Process_Attach){
Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL, 0, NULL);

}

Retornar a verdade;
```

```
}
```

Agora podemos construir esse código e injetá-lo no horror urbano para ver se ele funciona.

5.3.10 Convenção de Chamada

Quando nosso DLL for injetado, você notará que o jogo trava imediatamente ao iniciar e o seguinte erro ocorre:



Se removermos as chamadas para glDepthFunc do Code Cave, o jogo não trava mais. Parece que nossos ponteiros de função estão incorretos em alguns aspectos. Se olharmos para gl.h, veremos que glDepthFunc é definido como:

```
GLAPI void GLDEPTHUNC (GLenum func);
```

Na documentação da Microsoft sobre tipos de dados, vemos APIINTRY como referência ao WINAPI. Se olharmos para a entrada da WINAPI, descobriremos que ela é uma referência para _stdcall. Vamos tentar adicionar este prefixo ao ponteiro de função:

```
void(__STDCal*GLDEPTHUNC)(unsigned int)=NULL;
```

Construir este código pode resultar em um erro comum:

```
Erro C2440: '=': Não pode ser convertido de 'void(__CDCL*)(unsigned int)' para
'void(_StDCL*)(unsigned int)'
```

Isso pode ser resolvido mudando os personagens como fizemos antes:

```
GLDEPTHUNC=(void(__StDEPTHUNT)
```

A convenção de chamada controla como a função processa os parâmetros durante a chamada . Existem muitos tipos diferentes, mas para o nosso propósito, precisamos apenas saber que o Visual Studio usa __cdecl por padrão, enquanto o OpenGL usa __stdcall por padrão.

Com essa mudança, crie o código e injete-o novamente. Você notará que o terror urbano não está mais em colapso.

5.3.11 Verificação e contagem

Se você se juntar a um jogo, notará que agora pode ver entidades através das paredes. O único problema é que você pode ver muita coisa:

Em nossos hackers atuais, desativamos os testes de profundidade de todos os elementos desenhados na tela, incluindo paredes e escadas. Idealmente, só queríamos desenhar jogadores através de paredes. Para fazer isso, temos que filtrar elementos com os quais não nos importamos.

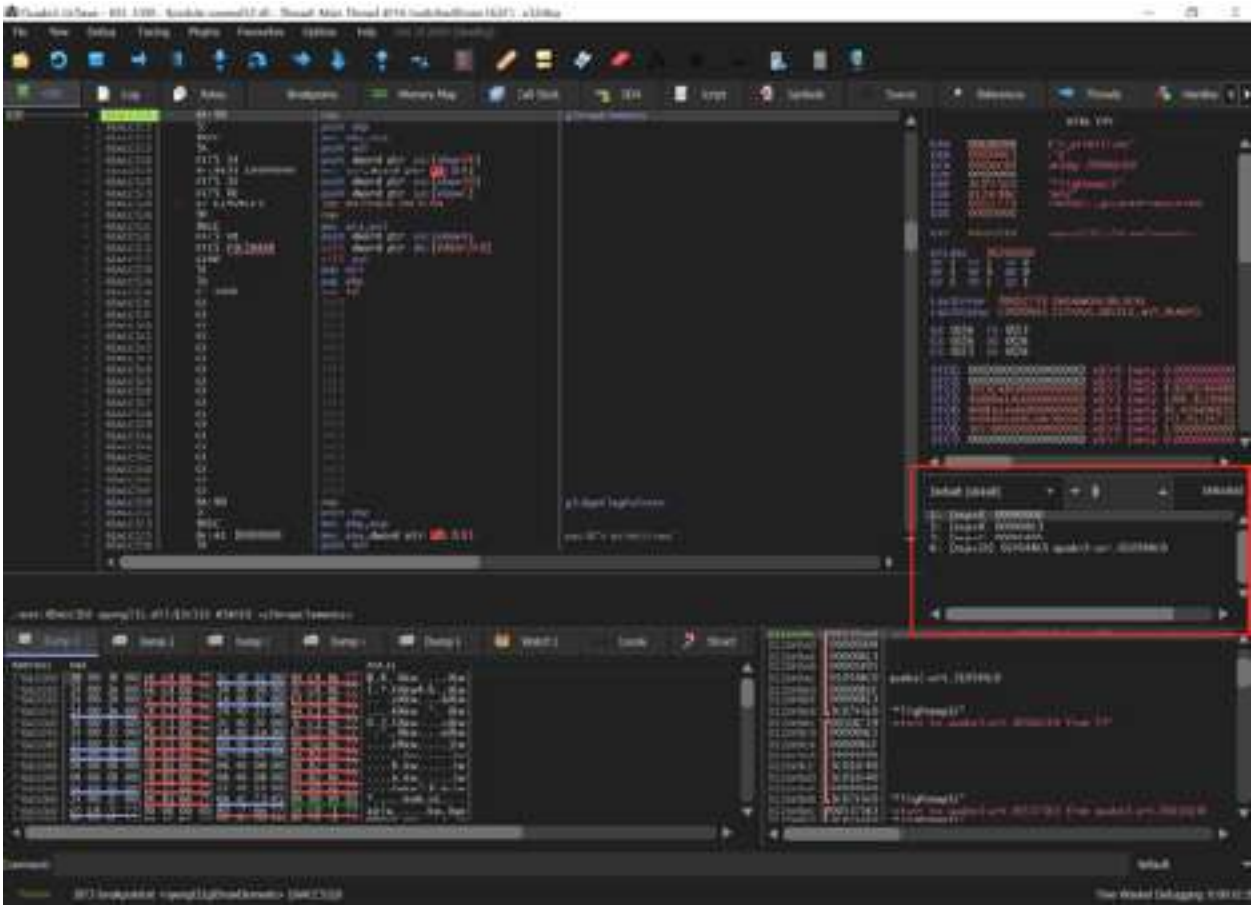
GLDRAWEMENTS tem as seguintes definições:

```
void glDrawElements (modo GLenum, contagem
GLenum, tipo GLenum, índice const void*);
```

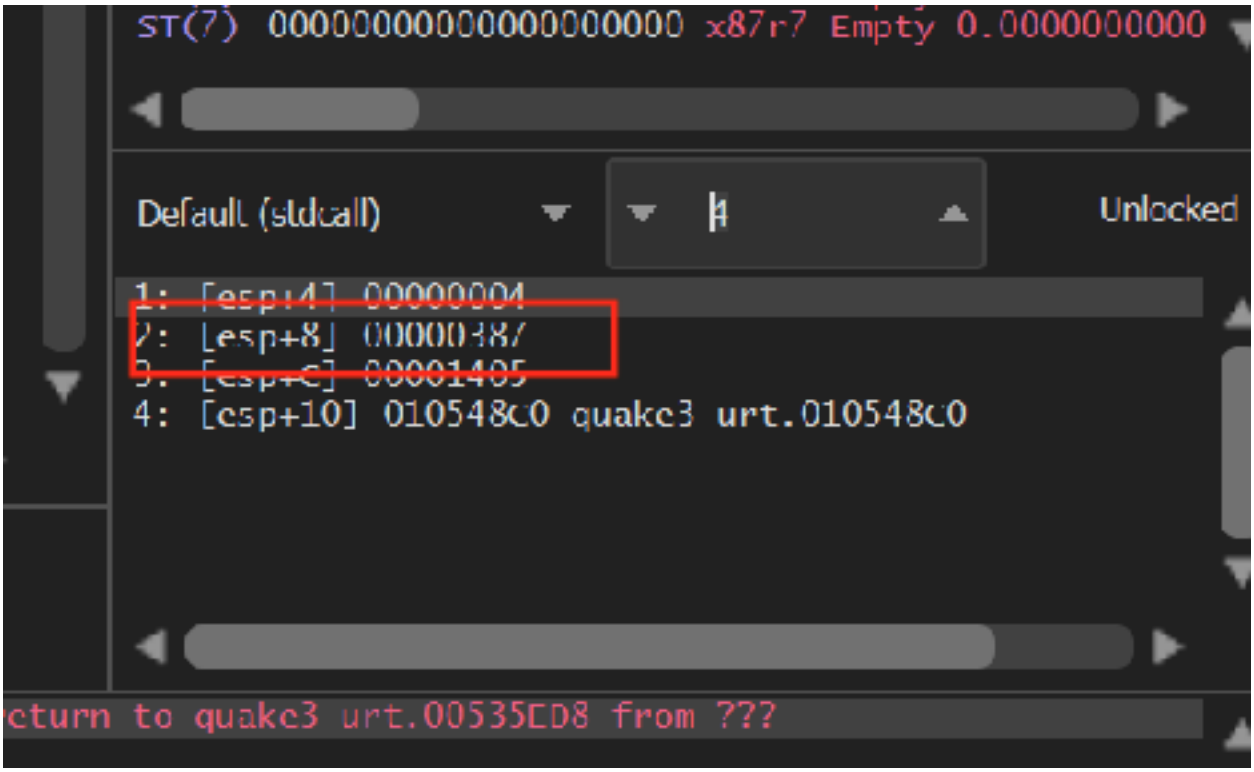
O parâmetro de contato especifica o número de elementos ou vértices a serem renderizados. Objetos mais detalhados terão mais vértices. Por exemplo, o modelo do jogador terá mais detalhes (nariz, mãos, dedos, etc.) do que o chão. Ao garantir que o parâmetro de contato seja um determinado valor, podemos filtrar elementos que não queremos ser exibidos na parede.

Sabemos que esse parâmetro estará na pilha quando nosso gancho pular para. Para recuperar sua localização exata, podemos injetar nosso DLL e definir um ponto de interrupção no glDrawElements. Quando executamos o código em uma única etapa, podemos identificar onde a caverna de código está na pilha quando ela é chamada.

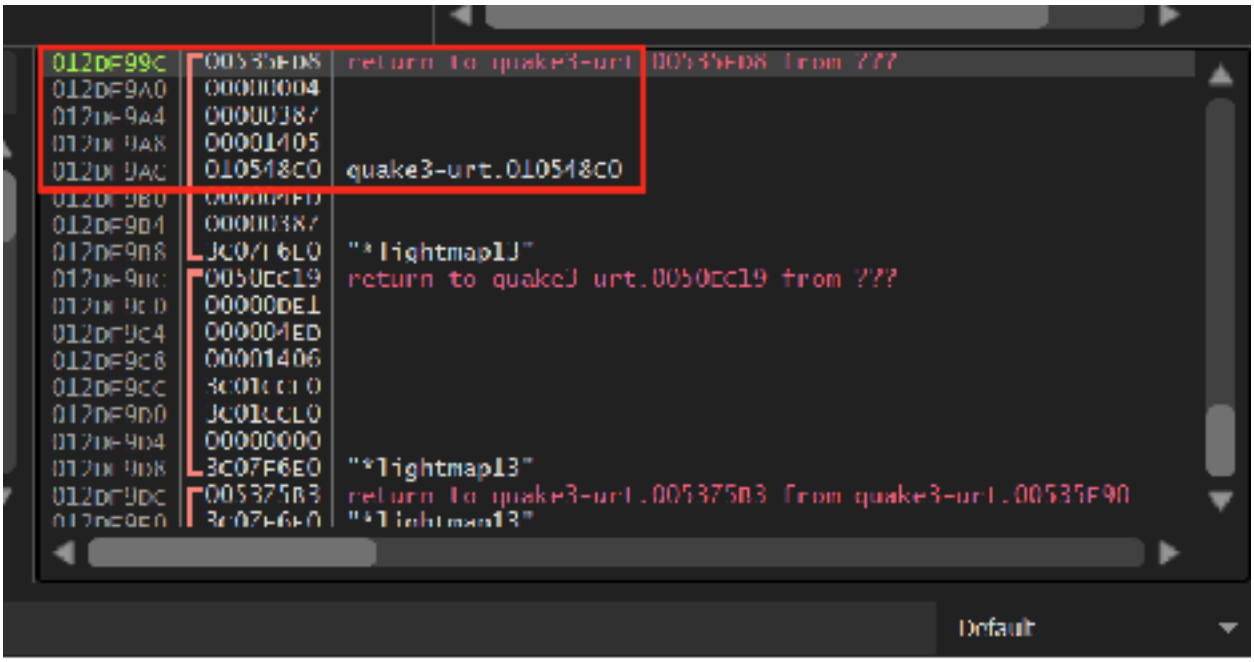
Um recurso do x64dbg é a capacidade de visualizar os parâmetros atuais na pilha de uma maneira semelhante à de passar parâmetros na linguagem C. Este recurso está localizado abaixo do painel que exibe o valor do registro:



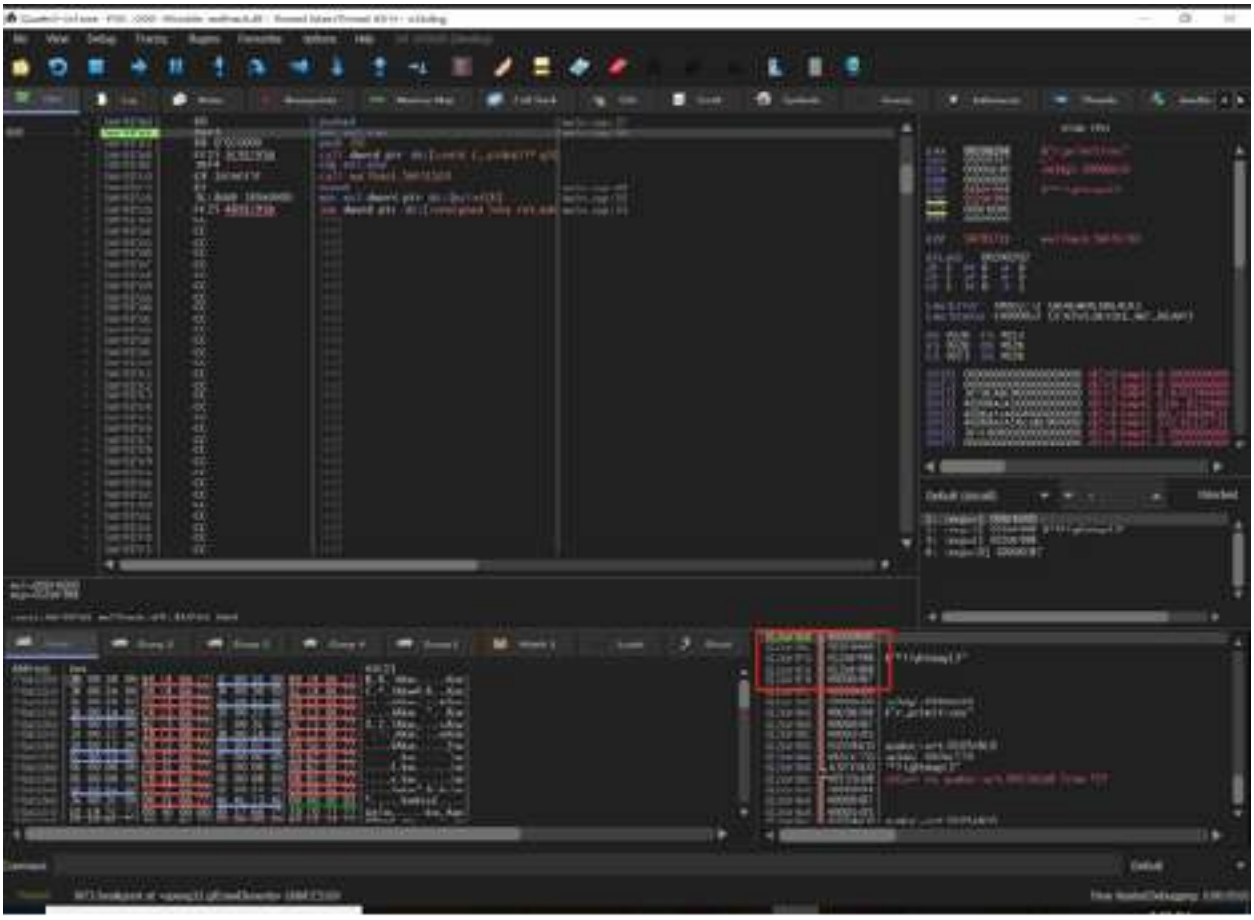
x64dbg não pode ser preenchido automaticamente, então você precisa definir a convenção de chamada e o número de parâmetros. Se acionarmos pontos de interrupção no glDrawElements várias vezes, podemos ver que [esp+8] é o único valor que parece ter mudado. Podemos assumir que ele salva o valor do parâmetro de contato no início da função:



Se olharmos para o painel da pilha no canto inferior direito, podemos ver como essas informações são representadas na pilha. Por padrão, o topo da pilha (esp) sempre aparecerá no topo da janela:



Continue executando a função em um único passo e depois pule para nossa caverna de código. Após o comando Pusha em nossa caverna de código, verifique a pilha novamente:



Neste ponto, podemos ver que o parâmetro de contato está em esp+0x10. Podemos citar esse valor na Caverna de Código para recuperar a contagem atual do elemento que está sendo apresentado. No primeiro bloco asm, após a instrução Pusha, podemos obter o valor esp+0x10 e armazená-lo em uma variável local:

```
Contagem DWORD = 0;
...
_asm{
Pushad
mov eax, dword ptr ds: [esp+0x10] Contagem
de mov, eax
Ppad
Pushad}
```

Agora temos uma variável local, conto, que salva o valor do conto transferido para o glDrawElements. Podemos então comparar esse valor com uma linha de base e desativar o teste de profundidade somente quando essa linha de base for excedida. Se não ultrapassarmos isso, reativaremos os testes de profundidade. O valor de GL_LEQUAL(0x 203) é o mesmo que o valor de GL_ALWAYS. Agora, usaremos 500 como valor de base:

```
Se (contar > 500){
(*glDepthFunc)(0x207);}

Caso contrário{
(*glDepthFunc)(0x203);}
```

Se construirmos e injetarmos isso, veremos que nossa visão agora está muito mais limpa, com apenas alguns elementos aparecendo através das paredes:

5.3.12 Planos de corte

Agora estamos filtrando muitos elementos, mas estamos enfrentando o problema de não haver modelos de jogadores. Em vez disso, só podemos ver suas armas e efeitos sanguíneos através das paredes:

Se ativarmos a visualização em terceira pessoa, nosso modelo de jogador também ficará invisível. A única coisa em que nosso modelo de player aparece é se ligarmos sem clipe e voarmos para fora dos limites. Isso provavelmente ocorre porque nosso modelo de jogador foi desenhado primeiro ao renderizar a cena e depois outros elementos do nível foram desenhados nela. Quando desativamos o teste profundo, todas essas entidades são desenhadas na parte superior do player.

```
draw_player();
draw_guns();
draw_doors();
Draw_LEV_Walls();
```

Para forçar os jogadores a serem desenhados sobre esses elementos, podemos usar a função glDepthRange. Esta função define o plano de corte próximo e o plano de corte distante da cena. Um plano de corte é um plano que se estende através da cena do jogo e corta (ou remove) qualquer entidade atrás dela. Ao definir esses valores para 0, os planos se cruzarão, fazendo com que todos os elementos sejam desenhados no mesmo plano e "compitam" pelo espaço de renderização. Isso causará alguns piscadores, mas o modelo do jogador aparecerá através da parede.

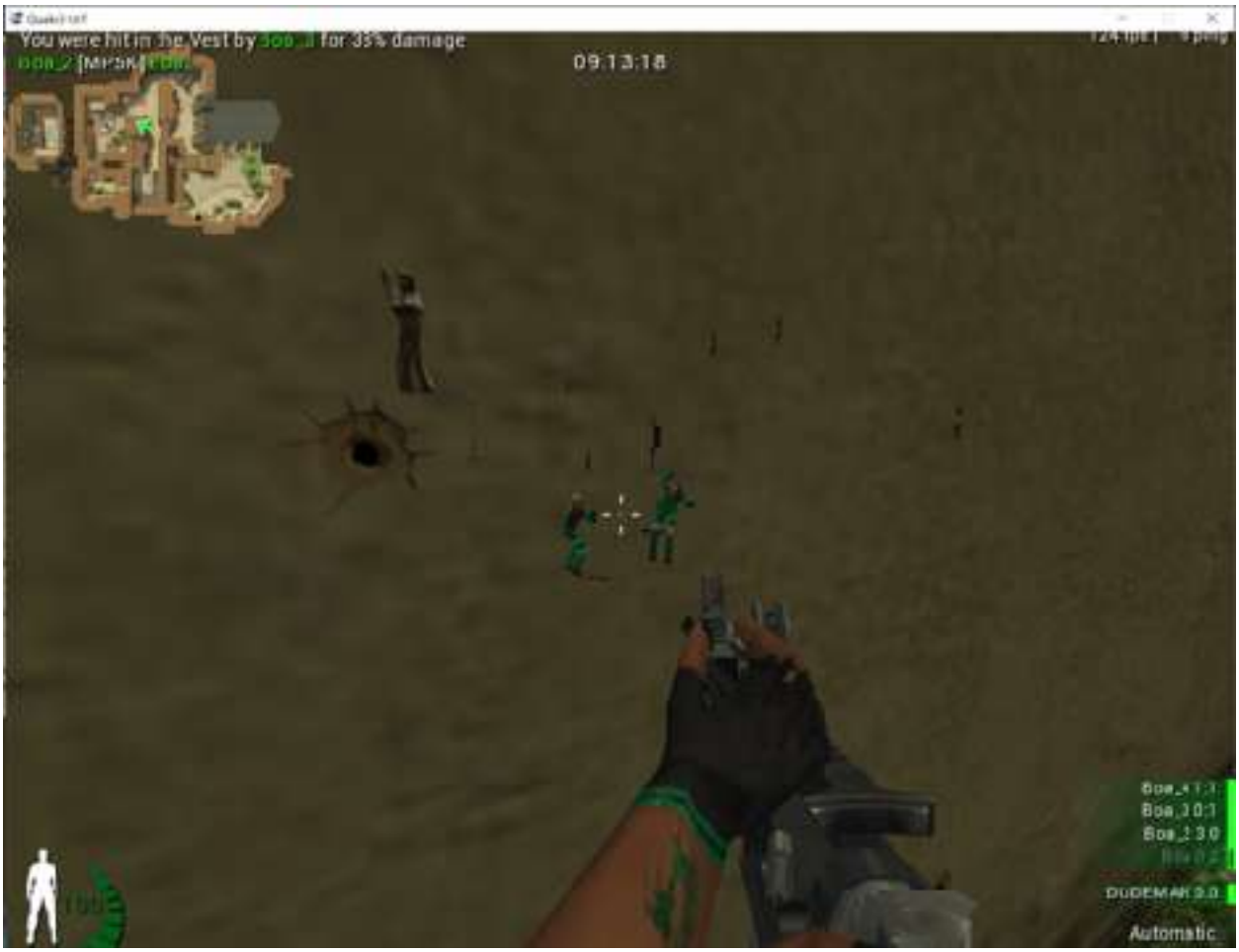
Podemos criar um ponteiro de função para esta função, assim como usamos para glDepthFunc. As únicas mudanças que precisamos fazer são os parâmetros:

```
void (__STDCal*GLDEPTHRGE) (double, double)=nul;
...
Gldepathrange=(void __stdcal*) (double, double) Gldepathrange (
Opengldepathrange, "Gldepathrange");
```

Podemos então chamar a função no mesmo local que a função de profundidade muda. O valor padrão para esses planos é (0,1) e, se a contagem for muito baixa, redefinimos esse valor.

```
Se (contar > 500){
(*GLDEPTHRGE) (0,00,0); (*
glDepthFunc) (0x207);
}
Caso contrário{
(*GLDEPTHRGE) (0,0,1,0); (*
glDepthFunc)(0x203);
}
```

Dessa forma, o modelo do jogador agora aparecerá através da parede, indicando que nosso wallhack foi bem-sucedido:



O código completo para este capítulo está disponível no Apêndice A.

5.4 Campos (OpenGL)

5.4.1 Metas

Nosso capítulo tem como objetivo o terror urbano 4.3.4.

5.4.2 Identificação

Nosso objetivo neste capítulo é criar um hacker chams, um hacker que pinta todos os jogadores com cores brilhantes. Podemos fazer isso vinculando a biblioteca gráfica do jogo e modificando seu código para que todos os modelos do jogador sejam renderizados em cores brilhantes em vez de texturas.

5.4.3 Compreensão

Quando as entidades são renderizadas na tela, elas são apenas polígonos preenchidos . Para que essas entidades tenham um efeito visual (como olhos, camuflagem ou cabelo), a textura deve ser aplicada a polígonos. Essas texturas são imagens especialmente formatadas que, quando aplicadas a entidades, circundam entidades. Por exemplo, a textura do barril de óleo em Urban Horror se parece com:

Quando aplicado a um modelo de barril circular, ele o circunda. É assim que a textura 2 D é aplicada a modelos 3D.

Para criar um chams hack, modificaremos esse processo de renderização. Após a renderização do polígono, desativaremos a textura no OpenGL. Quando a textura for desativada, o OpenGL retornará ao array de iluminação (ou cor) especificado pelo jogo . Se também o desativarmos, o OpenGL retornará à cor especificada no final da chamada GCLOR. Se definirmos nossas próprias cores e renderizarmos a entidade, podemos fazê-la aparecer como uma cor sólida brilhante, como vermelho.

5.4.4 Ponteiro de função de textura

Para facilitar nosso desenvolvimento, construiremos com base no OpenGL wallhack, criado no capítulo anterior. Para isso, criamos ponteiros de função para duas funções relacionadas ao teste de profundidade: GldePTHUNC e glDepthRange. Para desativar e habilitar texturas, precisamos criar ponteiros de função apontando para quatro funções adicionais:

- **glEnable**
- **GLDISEBLE**
- **GlenableClimentation Station**
- **GLDISBLIENTATE**

Planejamos usar glEnable e GLDISALL para habilitar e desativar GL_COLOR_ MATERIAL. Além disso, precisamos chamar GlenebleClimentation Station e GlenebleClimentation Station para habilitar e desativar GL_COLOR_ARRAY e GL_ Texture_Coor_Array. Ativaremos e desativaremos todos esses elementos para garantir que o OpenGL retorne ao modo onde podemos definir cores.

Para definir a cor após concluir essas etapas, também precisamos criar um ponteiro de função apontando para GCLOR. glColor tem muitos formulários que permitem que você passe diferentes tipos de parâmetros. Qualquer uma dessas funções funciona, mas para nossos hackers, usaremos a versão glColor4f, glColor, que aceita quatro números de ponto flutuante (valores decimais permitidos): um representa vermelho, verde, azul e alfa. O ponto flutuante alpha é responsável por controlar a opacidade da cor.

Podemos declarar esses ponteiros de função diretamente abaixo dos ponteiros GLDEPTHUNC e glDepthRange:

```
void (__stdcall*glColor4f) (float, float, float, float)=NULL;
void (__STDCal*Glennable) (unsigned int)=NULL;
```

```
void(__STDCal*GLDISLE) (unsigned int)=NULL;
void (__STDCal*GlenableClienteStation) (unsigned int)=NULL;
void (__stdcall*GLDISLEINTATE) (UnsigNET)=NULL;
...
...
glColor4f=(void(__stdcall*) (FLOT, FLOT, FLOT,
float) GetProcadodres (openGLHandle, "glColor4f"); Glenanable=(void(__
StDCal*) (unsigned int)) Glenanable (OpenGlenanable, "Glenanable");

glDisable=(void(__StDSTAL*)(unsigned int)) GetProcadernets (Opengldisable
, "glDisable");
Glenglândalo=(void(__stdcal*)(unsigned int)) Glenglândalo (
Openglândalo, "Glenglândalo"); Gldisable Climentation =(void
(__stcal*)(unsigned int)) Gldisable Climentation (
OpenGldisable Climentation, "Gldisable Climentation");
```

5.4.5 Caverna de Código glDrawElements

Em nossa caverna de código, construímos lógica para exibir o modelo através da parede se a contagem do modelo for maior que 500. Expandiremos esse código e os coloriremos. Primeiro, desativaremos GL_COLOR_ARRAY e GL_TEXTURE_COORD_ARRAY. O jogo usa esses estados do cliente para informar a OpenGL que o jogo quer mapear matrizes de texturas e cores (para iluminação) para polígonos. Para aplicar cores estáticas ao modelo, precisamos dizer à OpenGL que não estamos usando esses recursos. Como esses são considerados status de cliente, usaremos o GLDISLEINTATE para desativá-los. Podemos partir de gl.h:

```
Se (contar > 500){
...
(*GLDISLEITATE)     (0x8078);   (*
GLDISBLILENTATE)(0x8076);
```

Em seguida, habilitaremos GL_COLOR_MATERIAL e definiremos nossa cor para vermelho. Para todos os valores, glColor4f toma um valor entre 0 e 1. Se quisermos vermelho, definimos o valor vermelho para 1 e o alfa para 1. Entretanto, se deixarmos o verde e o azul em 0, o vermelho que terminaremos será escuro e suave. Para torná-lo vibrante, definimos esses valores para 0,6. Em C++, adicionar um f no final de um número faz com que o número seja interpretado como um ponto flutuante:

```
(*glEnable)(0x0B57);
(*glColor4f)(1,0F, 0,6F, 0,6F, 1,0F);
```

Por fim, assim como em nosso wallhack, desativaremos essa coloração quando a contagem do modelo for inferior a 500. Para isso, ativaremos GL_COLOR_ARRAY e Gl_Texture_CORD_Array e então desative Gl_CORD_Matérial. Por fim, chamaremos glColor novamente, desta vez definindo a cor para branco puro. Isso não é absolutamente necessário para o horror urbano, mas para alguns jogos isso impedirá que a cor do efeito seja destruída ao usar a cor que você definiu anteriormente:

```
(*GlenableClimentation Station)(0x8078);
(*GlenableClimentation Station)(0x8076);
(*GLDISBLE)(0x0B57);
(*glColor4f)(1,0F, 1,0F, 1,0F, 1,0F);
```

Depois de concluir isso, você poderá injetar DLL no jogo e verá todos os modelos aparecendo em vermelho vivo através das paredes:

O código-fonte completo deste hacker pode ser encontrado no Apêndice a.

5.5 Robô de gatilho

5.5.1 Objetivos

Nosso objetivo neste capítulo será o Game Assault Cube 1.2.0.2, pois ele tem uma maneira simples de criar robôs e desativar seus movimentos. No entanto, a mesma técnica se aplica a qualquer FPS que exiba o nome do jogador quando você passa um rato sobre ele.

5.5.2 Identificação

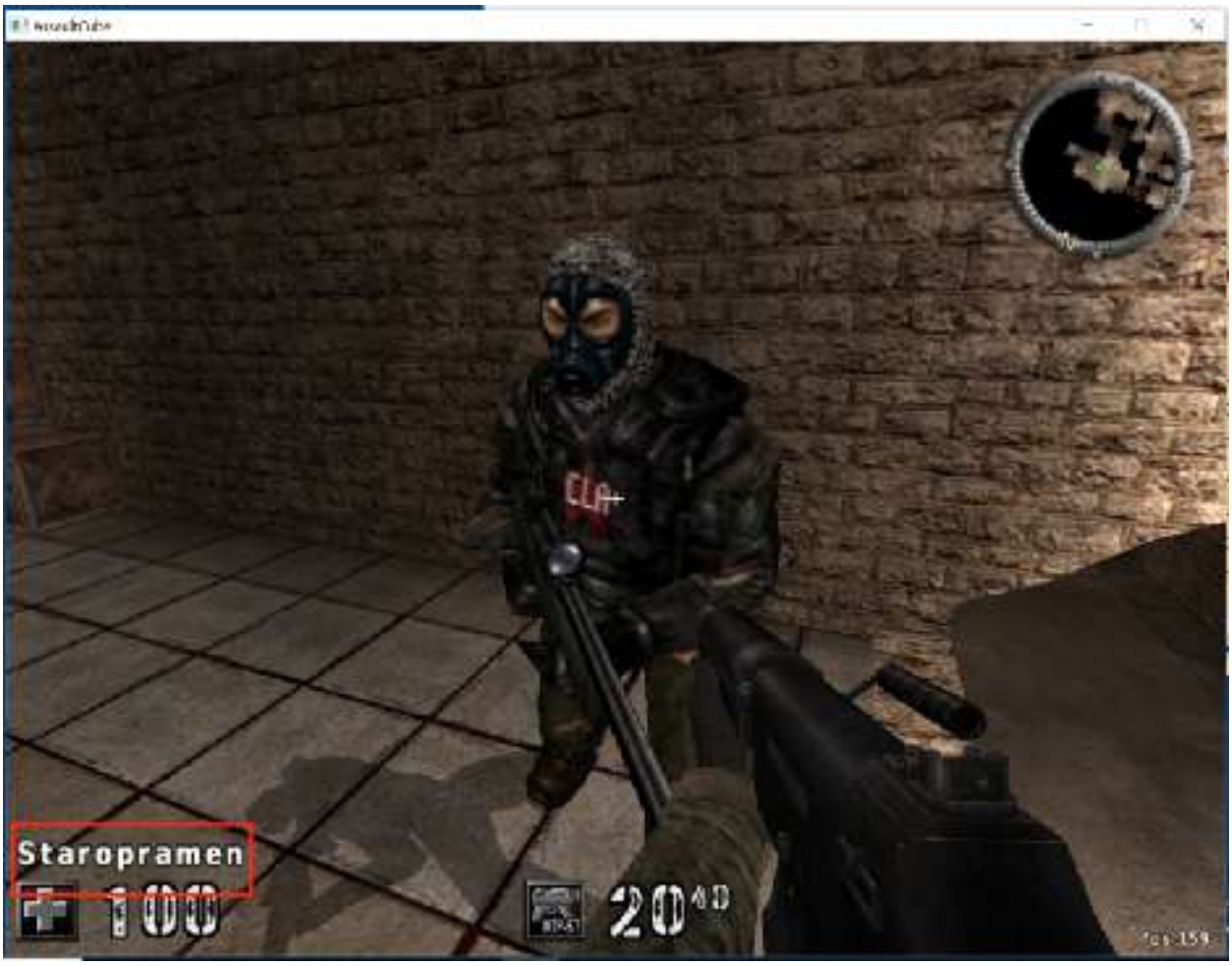
Nosso objetivo neste capítulo é criar um robô gatilho, um tipo de hacker que aciona automaticamente sempre que vemos outro jogador.

5.5.3 Injeção de DLL

Embora os capítulos a seguir possam ser concluídos usando a tecnologia AppInit discutida nos capítulos anteriores, usar um injetor DLL acelerará muito o tempo de desenvolvimento. A partir deste ponto, o restante do livro assumirá que você está usando uma seringa. O Capítulo 7.1 discute a questão da criação de um injetor DLL. Injetores de DLL universais também podem ser encontrados on-line.

5.5.4 Compreensão

Para escrever o Trigerbot, precisamos calcular onde nossos jogadores estão assistindo e determinar se estamos assistindo a outro jogador. Felizmente, esse recurso já está disponível na maioria dos códigos dos jogos, e ele exibe uma marca quando você paira o mouse sobre o jogador ou muda a linha cruzada para uma cor diferente. O Assault Cube exibe uma etiqueta de nome, conforme mostrado no canto inferior esquerdo:



Se encontrarmos o código responsável por exibir esse texto, podemos enganá-lo e escrever um código personalizado para enviar um clipe do mouse.

5.5.5 Código de Localização

A maneira como o código é localizado dependerá de como o jogo reage ao jogador pairando. De modo geral, os jogos respondem de duas maneiras diferentes:

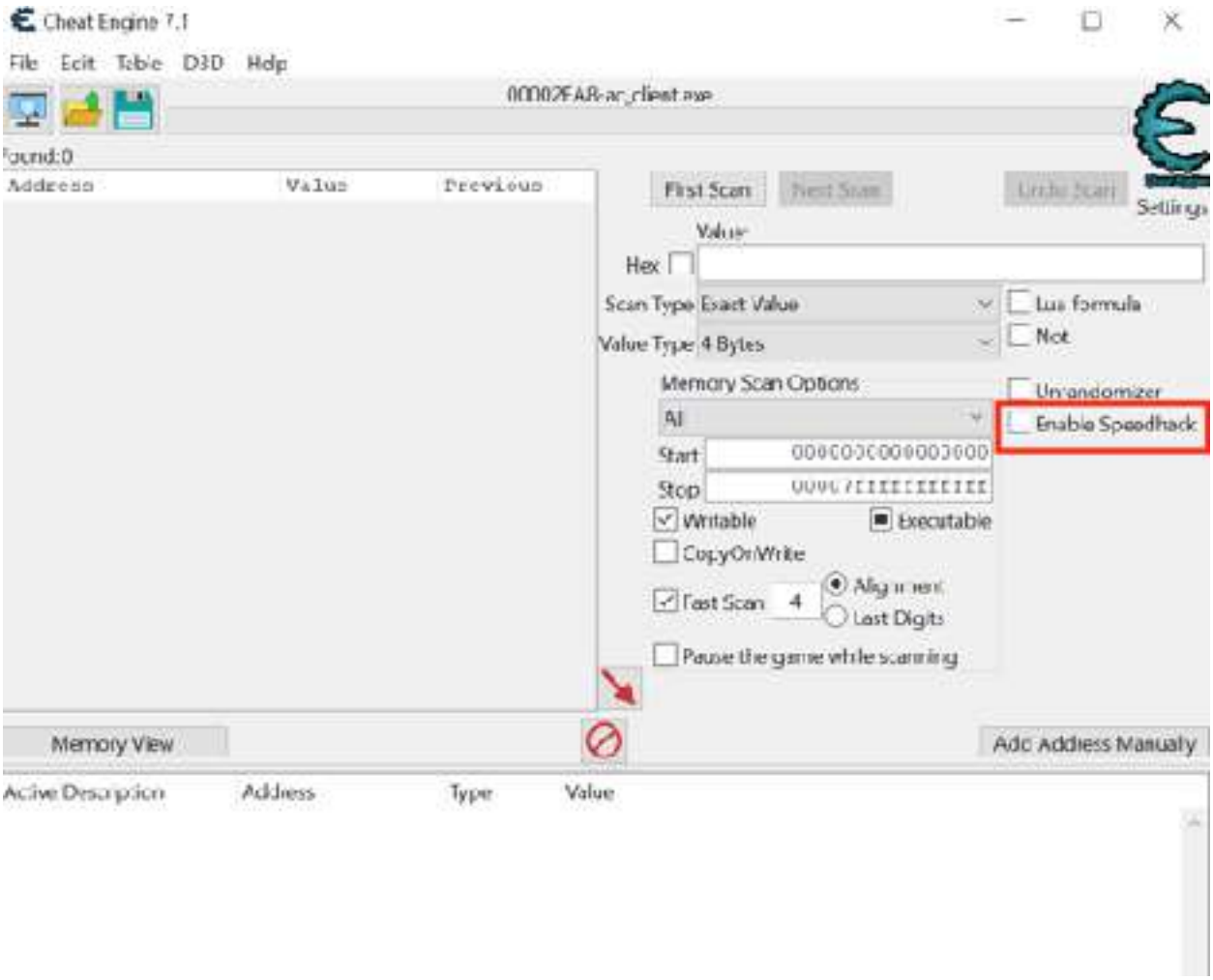
1. A linha cruzada mudará, seja o tamanho ou a cor.
2. O nome do jogador que estamos assistindo será exibido em algum lugar na tela.

No jogo de primeira reação, temos que pesquisar um valor sem olhar para o jogador e depois filtrar outro valor sem olhar para o jogador. Após filtragem suficiente, um valor será mantido. Quando o jogador não for olhado, esse valor geralmente é 0, enquanto 1

(ou um valor vinculado à posição do jogador na lista de entidades). Você pode então definir um ponto de interrupção neste endereço de memória e visualizar o código escrito nele.

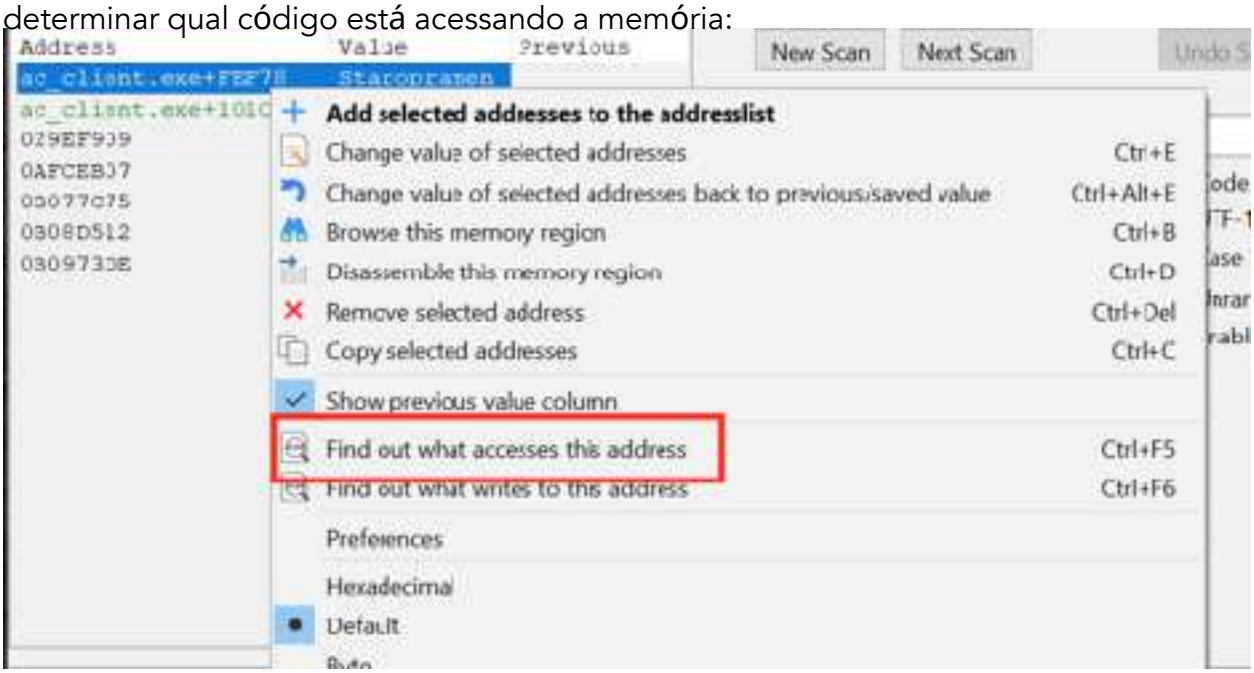
Outros jogos, como o Cubo de Ataque, exibem uma string que representa o nome do jogador. Nesses casos, podemos encontrar o nome de um jogador na memória e definir um ponto de interrupção ao acessar o nome. Quando pairamos o mouse sobre o player, esse ponto de interrupção deve aparecer no local responsável por determinar se estamos verificando o código do player.

Primeiro, certifique-se de que a opção de etiqueta de nome esteja habilitada nas configurações HUD. Em seguida, comece um novo jogo de competição de morte para um jogador com 8 robôs. No início do jogo, pressione a tecla "~" para abrir o console e executar o comando idlebots 1. Este comando proibirá o robô de se mover e atirar, facilitando a busca por informações que queremos. Em jogos onde o movimento do robô não pode ser desativado, você pode usar a função Enable Spectack do Cheat Engine para diminuir a velocidade do jogo e facilitar a pesquisa:

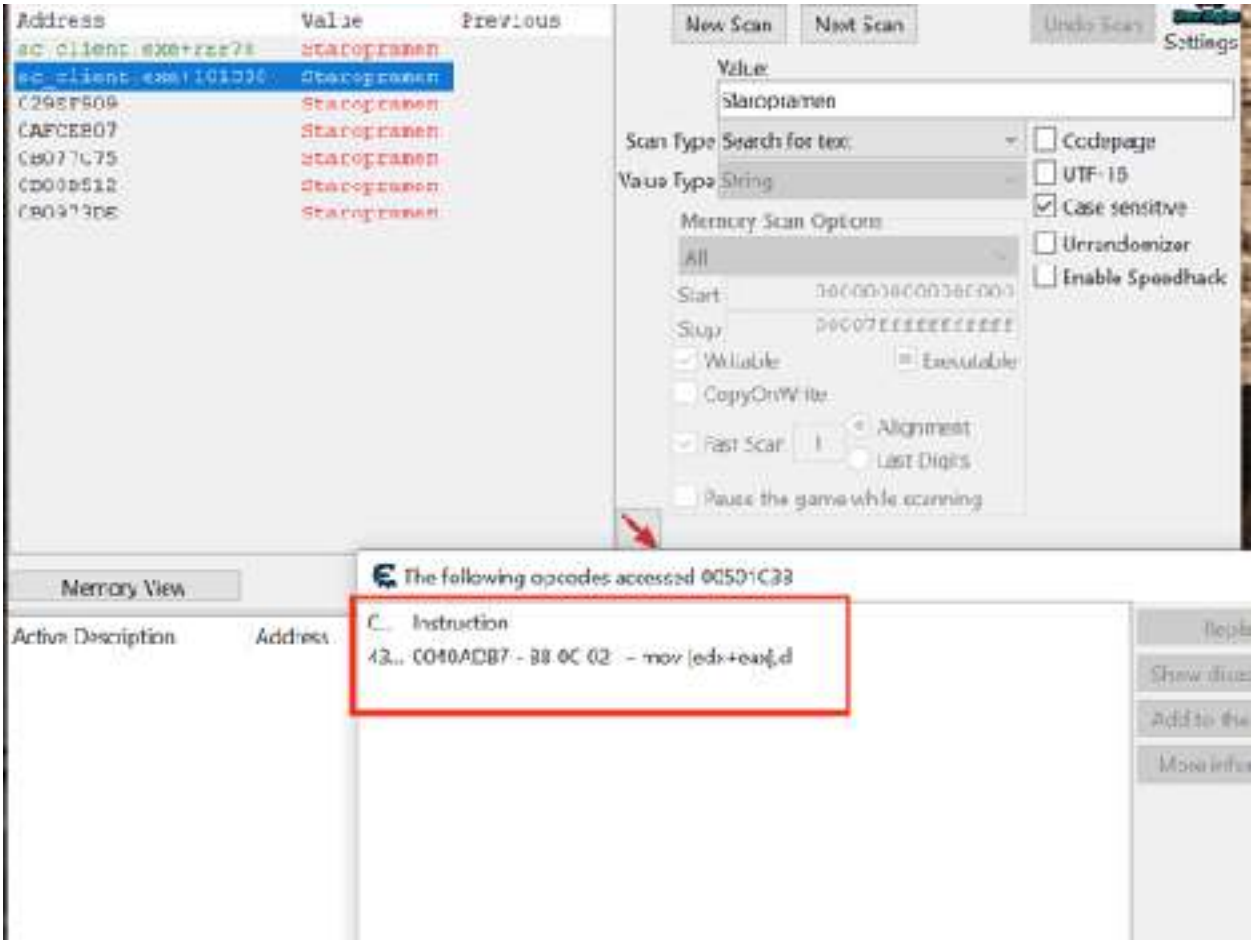


Quando o jogo começar, encontre um robô específico e anote seu nome. Pesquisar por este nome em um mecanismo de fraude deve retornar cerca de 10 resultados:

Em seguida, afaste seu olhar do robô para que seu nome não seja mais exibido. Para cada endereço identificado, use a opção "Encontrar o que acesso a este endereço" do mecanismo de fraude, que adiciona um depurador ao processo para determinar qual código está acessando a memória:



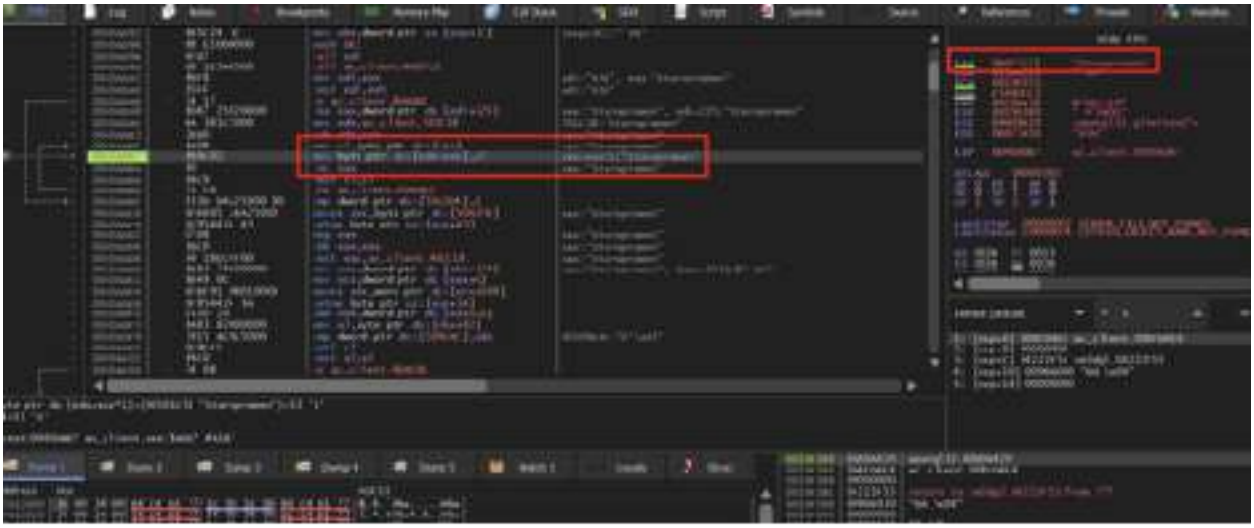
Para cada endereço, verifique o robô novamente para exibir uma etiqueta de nome. Estávamos procurando um endereço que só recebemos um grande número de visitas quando estamos observando robôs. Depois de navegar por vários endereços, você deve encontrar um endereço que só pode ser acessado ao visualizar o bot:



Depois de encontrar esse código, podemos desativar o mecanismo de trapaça e começar a reverter o código responsável.

5.5.6 Código de Localização Caverna

Abra o x64dbg e conecte-o ao Cubo de Ataque. No Cubo de Ataque, certifique-se de que a etiqueta de nome ainda esteja exibida. Navegue até o endereço que encontramos no mecanismo de fraude e coloque um ponto de interrupção no código. Deve sair imediatamente:



Você notará que estamos em um loop que carrega o nome do jogador no buffer. Vemos que esse ciclo só entra quando o je em 0x0040ADA6 não salta. A condição para esse salto é testar as instruções edi, edi. O próprio registro de teste compara seu valor com 0. A partir disso, podemos presumir que a etiqueta de nome só é exibida quando o edi não é 0. Podemos confirmar esse comportamento definindo pontos de interrupção nas instruções de teste edi e edi. Ao analisar um jogador, o edi tem um valor:

Entretanto, se não estivermos observando os jogadores, o edi permanecerá em 0:

Observando a chamada acima, podemos ver que o edi obtém seu valor da eax, que é o valor de retorno da chamada:

Como a chamada é de 5 bytes, usaremos uma caverna de código para cobrir a chamada em nosso próprio código.

5.5.7 Escreva Caverna de Código

A lógica por trás da nossa caverna de código será simples. Primeiro, executaremos a chamada que estamos ganhando. Em seguida, lemos o valor de eax em uma variável. Depois, leremos o valor da variável. Se estivermos assistindo a um jogador, usaremos a API SendInput para enviar um evento para baixo com o botão esquerdo do mouse para o jogo. Caso contrário, enviaremos um evento com o botão esquerdo do mouse para cima para o jogo . Precisamos usar esse método porque o SendInput define permanentemente o status da tecla. Se não enviarmos um evento com o botão esquerdo do mouse para cima, o botão do mouse parecerá pressionado e segurado.

Como discutimos acima, conectaremos a chamada em 0x0040AD9D. Faremos isso da mesma maneira que fizemos nos capítulos anteriores:

```
unsigned char*Hook_Localização=(unsigned char*)0x0040AD9D;
...
Projeto Virtuo* (void*)Hook_Locasione, 5. página _Express_Readwrite,
&old_protect);
*Hook_Localização=0xE9;
*(DWORD*)(hook_localização+1)=(DWORD)&codecavidade-((DWORD)hook_localização+5);
```

Em nossa caverna de código, primeiro chamaremos nosso método de reescrita e depois moveremos seu valor de retorno (eax) para a variável que declaramos:

```
DWORD or_call_adress=0x4607C0; DWORD
edi_value=0;

__declspec(naked)void codecave(){_
      asm{
            Chamar ori_call_address
            push
            mov edi_value, eax}
```

Em seguida, verificaremos o valor da variável edi_value para determinar se um evento com o mouse esquerdo para baixo ou para cima deve ser enviado:

```
Se(edi_valor!=0){
//Verificando o player
}
Caso contrário{
//Não assista ao tocador}
```

SendInput aceita um array de eventos de entrada que permite que você envie vários eventos. Isso pode ser útil se quisermos fazer vários movimentos ao mesmo tempo, como abrir fogo e recarregar. Neste capítulo, enviaremos apenas uma entrada, o evento mouse para baixo ou para cima:

```
Entrada ={0};
...
Se(edi_valor!=0){
Entrada. Tipo = mouse de entrada;
input.mi.dwFlags=MOUSEEVENTF_LEFTDOWN;
SendInput (1, &input, sizeout (input));
```

```
}
Caso contrário{
Entrada. Tipo = mouse de entrada;
input.mi.dwFlags=mouseeventf_lefutup;
SendInput (1, &input, sizeout (input));
}
```

Assim como nos capítulos anteriores, queremos restaurar o registro e retornar ao código original:

```
DWORDORI_Jump_Adress=0x0040ADA2;
...
_asm{
Rebote
JMPORI_Jump_Adress}
```

Se você estiver usando o injetor DLL do Capítulo 7.1, podemos fazer algumas pequenas modificações para aplicá-lo a este objetivo:

```
const char*DL_Path="C:\Users\\\IEUser\\\source\repos\Trigerberbot\Debug\\
Trigerberbot\dll";
...
Se(strcmp(((const char*)pe32 2. szExeFile, (const char*)L"ac_cliente.exe")==0
){
```

Com essas mudanças, podemos injetar DLL no Cubo de Ataque e pairar o mouse sobre o jogador. Quando passamos por um jogador, atiramos automaticamente.

O código completo para este capítulo está disponível no Apêndice A.

5.6 Amor

5.6.1 Objetivo

Nosso objetivo neste capítulo é atacar o Cubo Mágico 1,2.0,2.

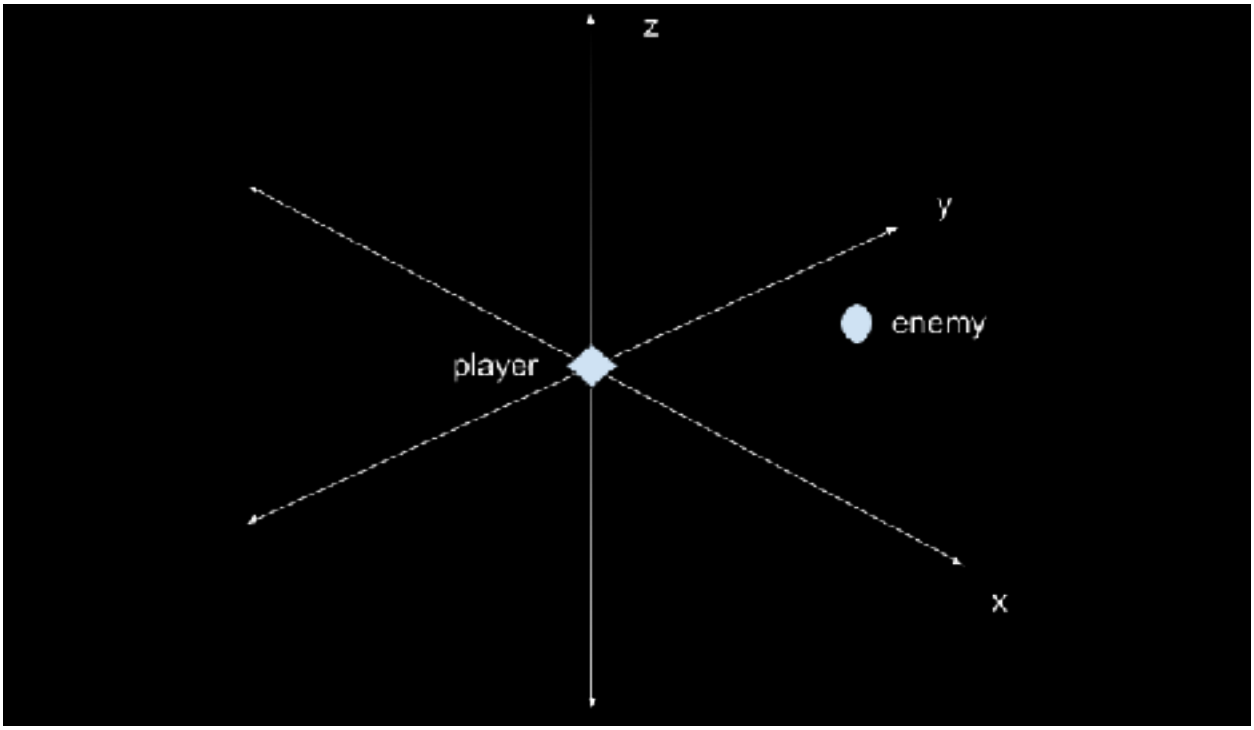
5.6.2 Identificação

Nosso objetivo neste capítulo é criar um aimbot, um hacker que mira automaticamente outros jogadores.

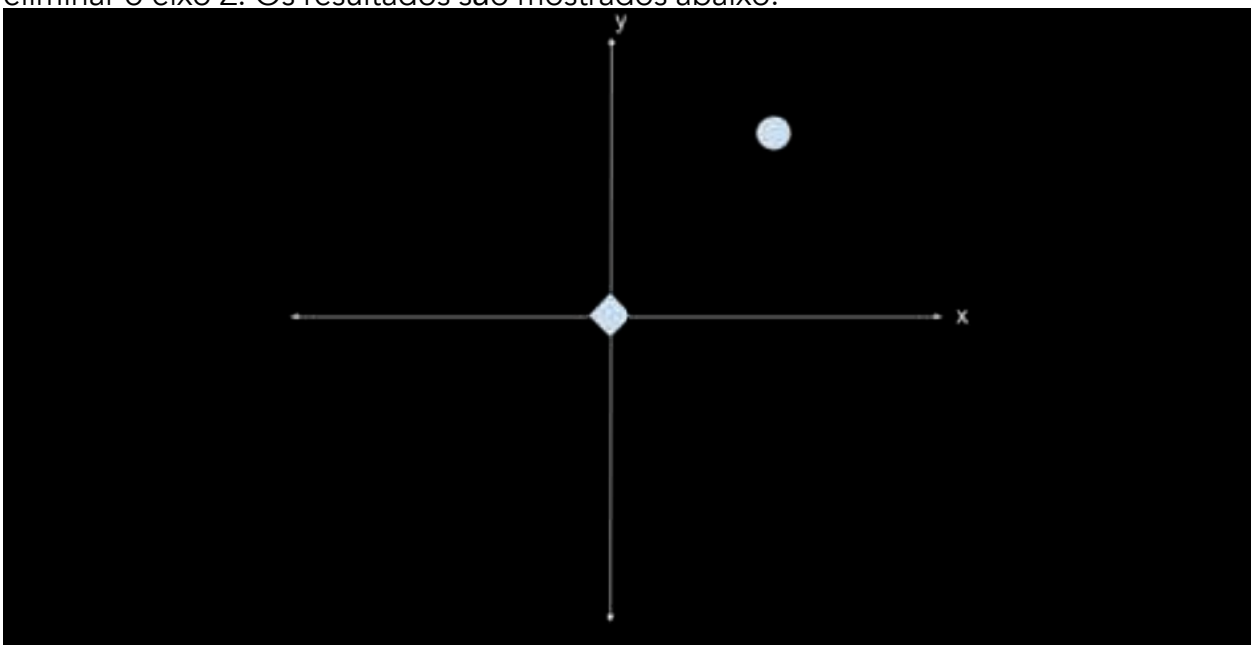
5.6.3 Compreensão

A base central do aimbot depende da trigonometria. Tomemos como exemplo as seguintes cenas do nosso jogo-alvo, Assault Cube:

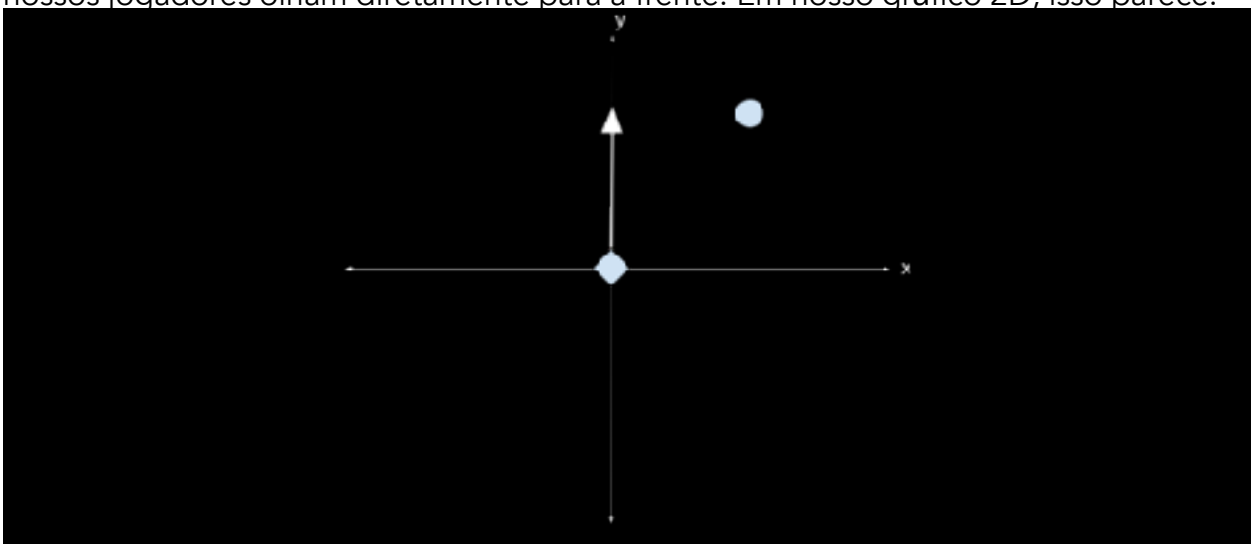
Concentre-se apenas em nossos jogadores e inimigos. Esta cena pode ser mapeada em gráficos 3D, conforme mostrado abaixo:



Para simplificá-lo, podemos convertê-lo em um gráfico 2D fixando o ângulo de visão e eliminando um dos eixos. Ao escolher uma perspectiva de cima para baixo, podemos eliminar o eixo Z. Os resultados são mostrados abaixo:

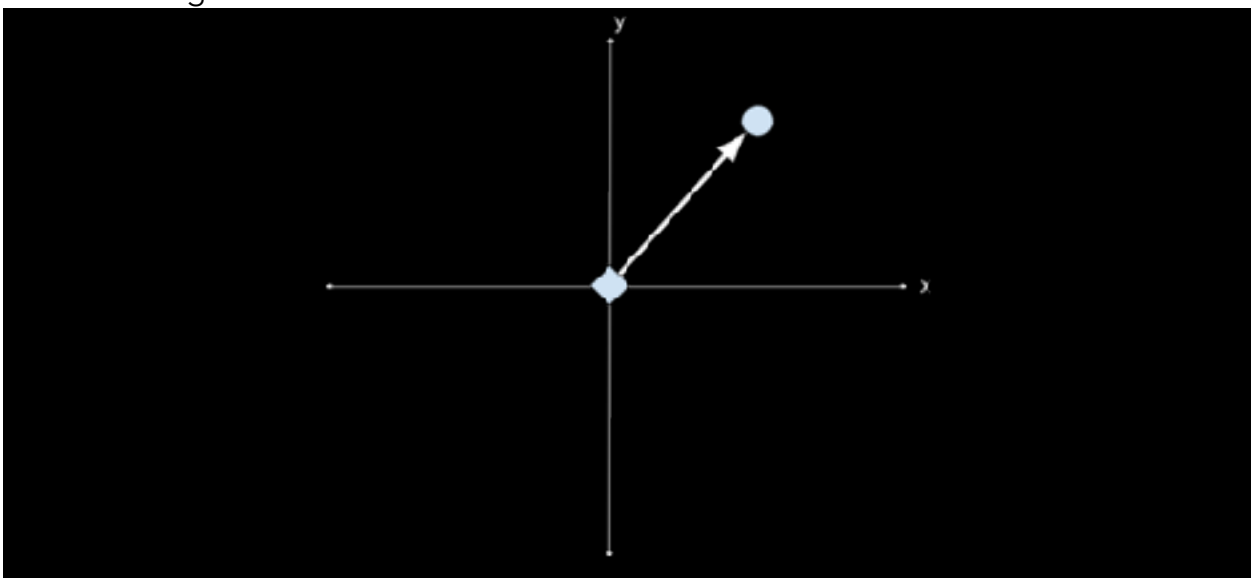


Cada jogo de tiro em primeira ou terceira pessoa permite que os jogadores olhem para a esquerda e para a direita para mirar. Por exemplo, em nossa primeira captura de tela, nossos jogadores olham diretamente para a frente. Em nosso gráfico 2D, isso parece:

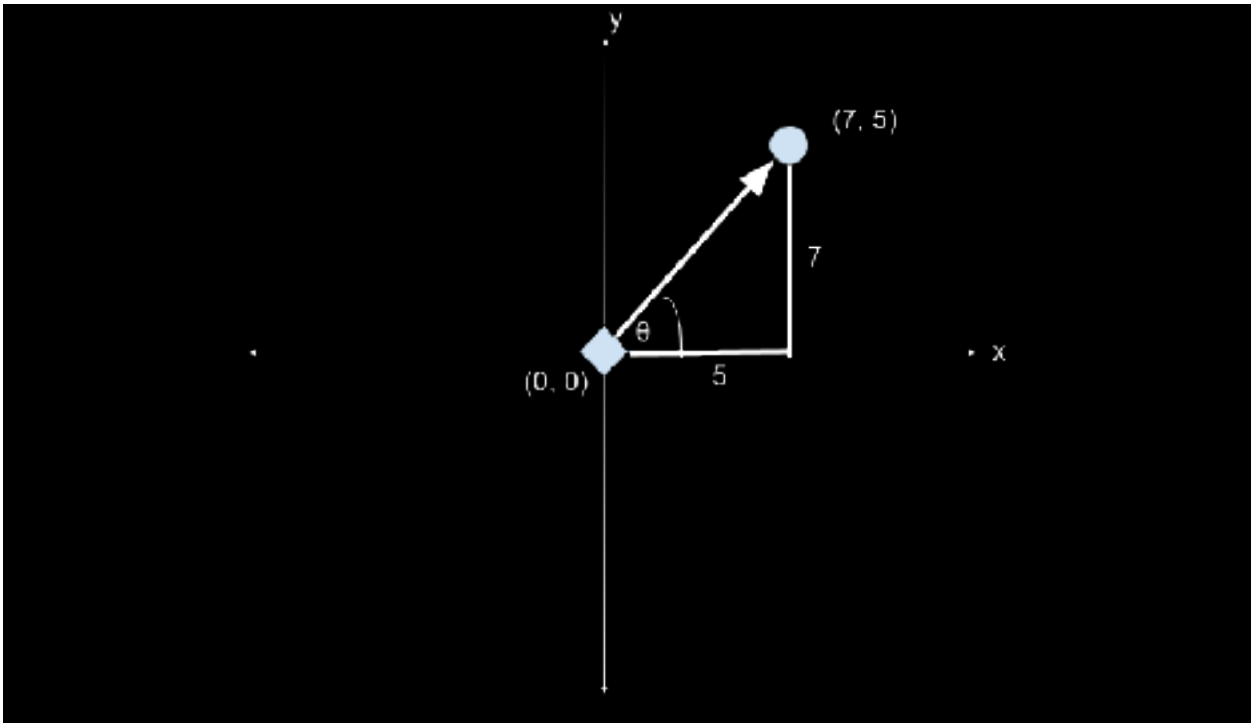


Se estamos olhando para um inimigo, como este:

Então nosso gráfico se tornará:



O jogo representa esse valor esquerdo e direito como um ângulo. Eles podem representar esse ângulo de várias maneiras, como vetores, radiais ou graus. Entretanto, para nosso exemplo atual, assumiremos que a perspectiva seja expressa em graus. Para criar um aimbot, precisamos encontrar uma maneira de calcular esse ângulo do inimigo. Podemos fazer isso criando primeiro um triângulo retângulo usando a posição do nosso jogador e a posição do inimigo:



Se conhecemos o valor, podemos usar a operação tangente para determinar a proporção entre a borda oposta (7 acima) e a borda adjacente (5 acima). Em nosso exemplo, temos

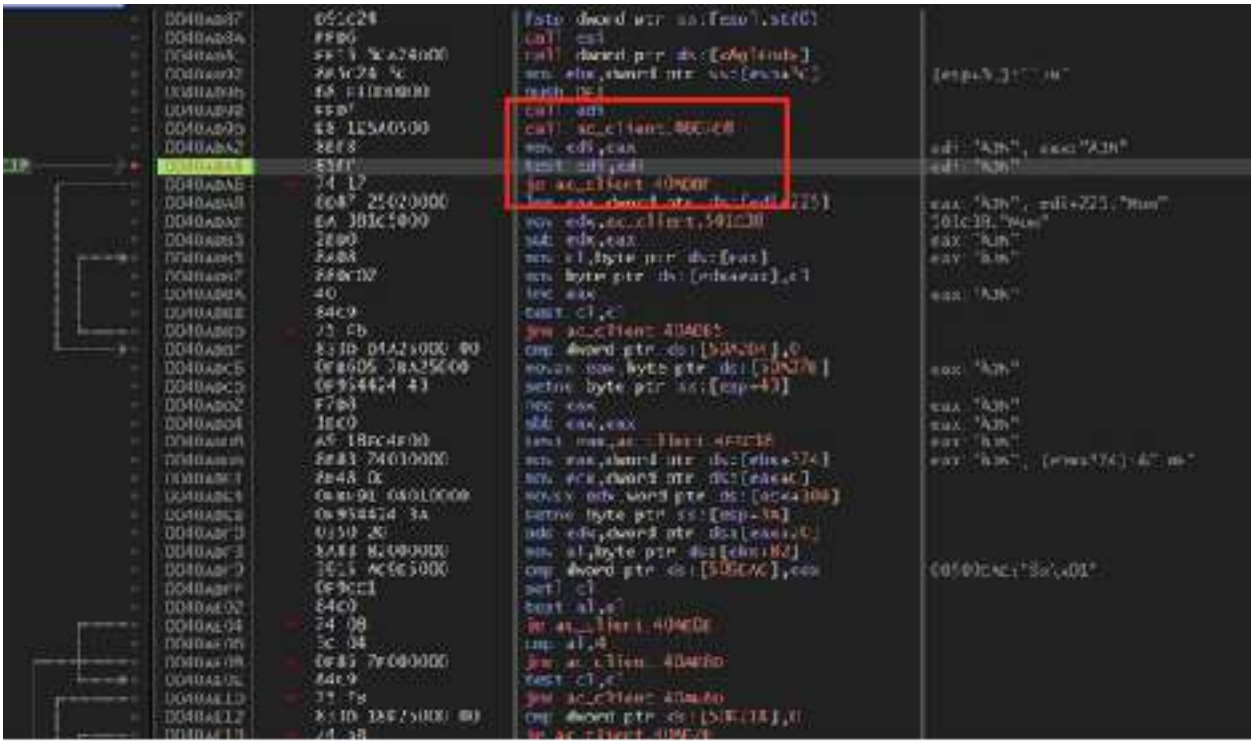
Lados opostos e lados adjacentes, e espero determinar. Para isso, podemos calcular tangente inversa ou tangente de qualquer maneira. De qualquer forma, cortar o general significa que precisamos definir o ângulo em que o jogador mira

No entanto, isso só apontará corretamente para a esquerda e para a direita. Para mirar para cima e para baixo, precisamos realizar operações semelhantes nos eixos Y e Z.

No entanto, antes de fazermos isso, precisamos encontrar onde o jogo armazenará inimigos. Depois, precisamos encontrar onde o jogo armazena os jogadores. Por fim, precisamos reverter a estrutura do player para localizar os membros X, Y e Z na estrutura, bem como os membros da perspectiva.

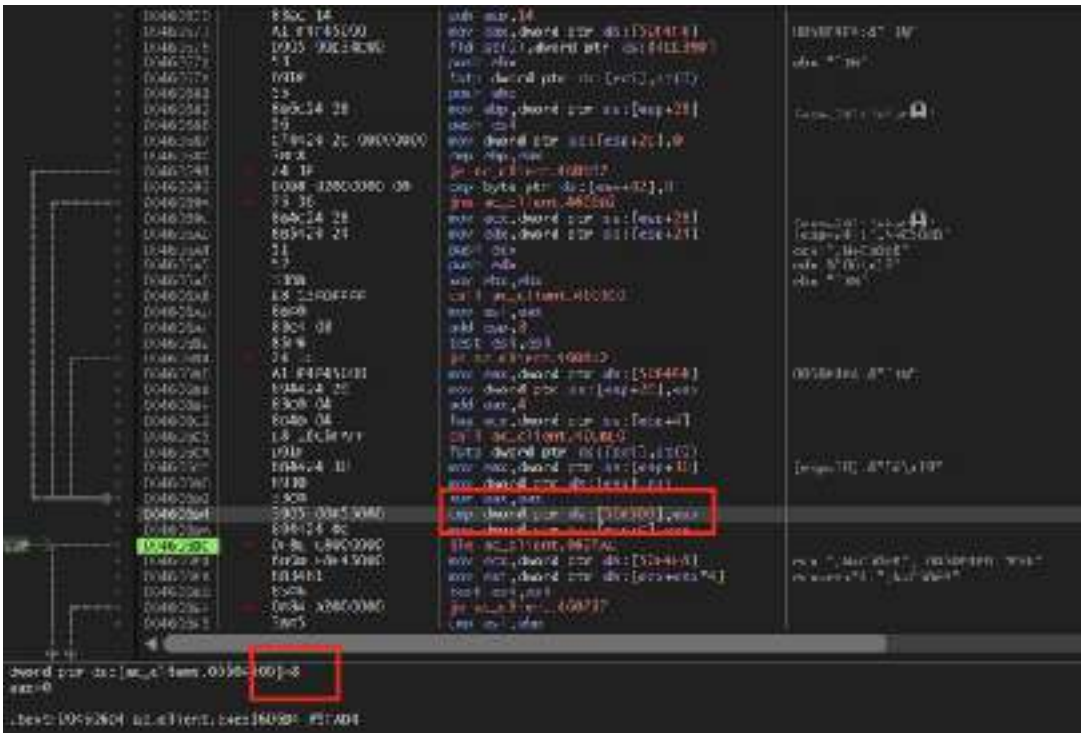
5.6.4 Localize o inimigo

Para localizar inimigos no jogo, primeiro crie um jogo com 8 robôs e deixe-os ociosos. Normalmente, o jogo armazena inimigos em uma lista e mantém uma posição estática nessa lista. No capítulo anterior, descobrimos o código do jogo responsável por exibir o nome do jogador quando você paira um rato sobre ele. Para fazer isso, o jogo deve incluir código na função, iterar os inimigos no jogo e pesquisar seus nomes. Aqui está o código que encontramos no capítulo anterior:

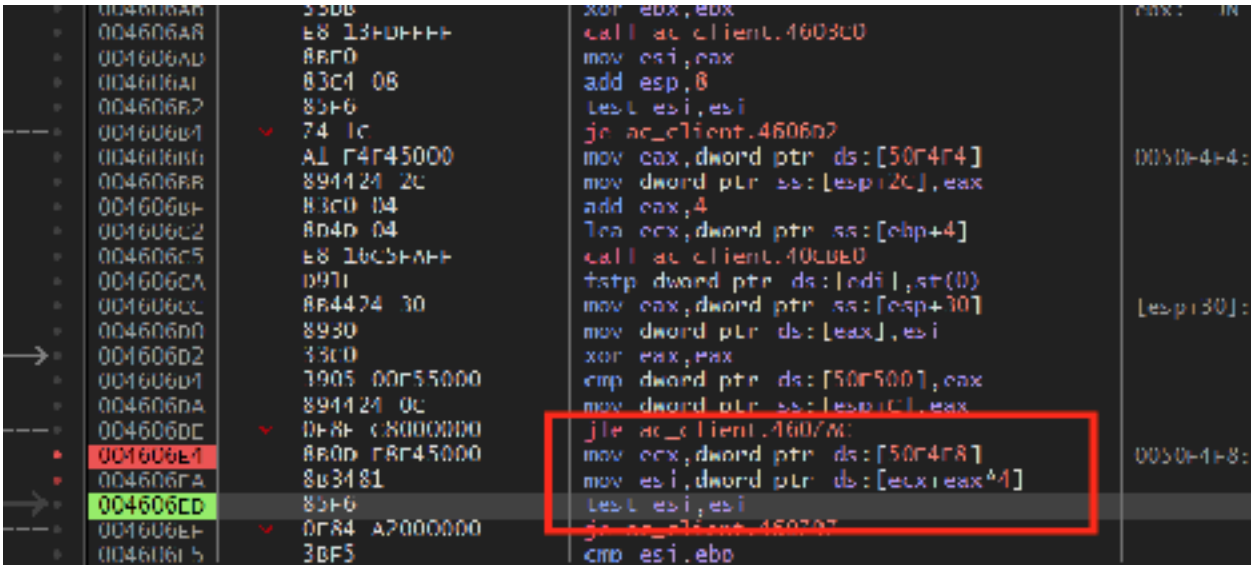


Ao inverter esse código, determinamos que a chamada 0x40ad9d para 0x4607c0 é responsável por carregar o player atualmente visualizado no eax. Se entrarmos nesta chamada, veremos que a chamada no final é responsável por obter esse valor:

Ao entrar nesta chamada, podemos ver que ela é bem longa, com muitos loops e condições . À medida que concluímos o código passo a passo, você notará as seguintes linhas:



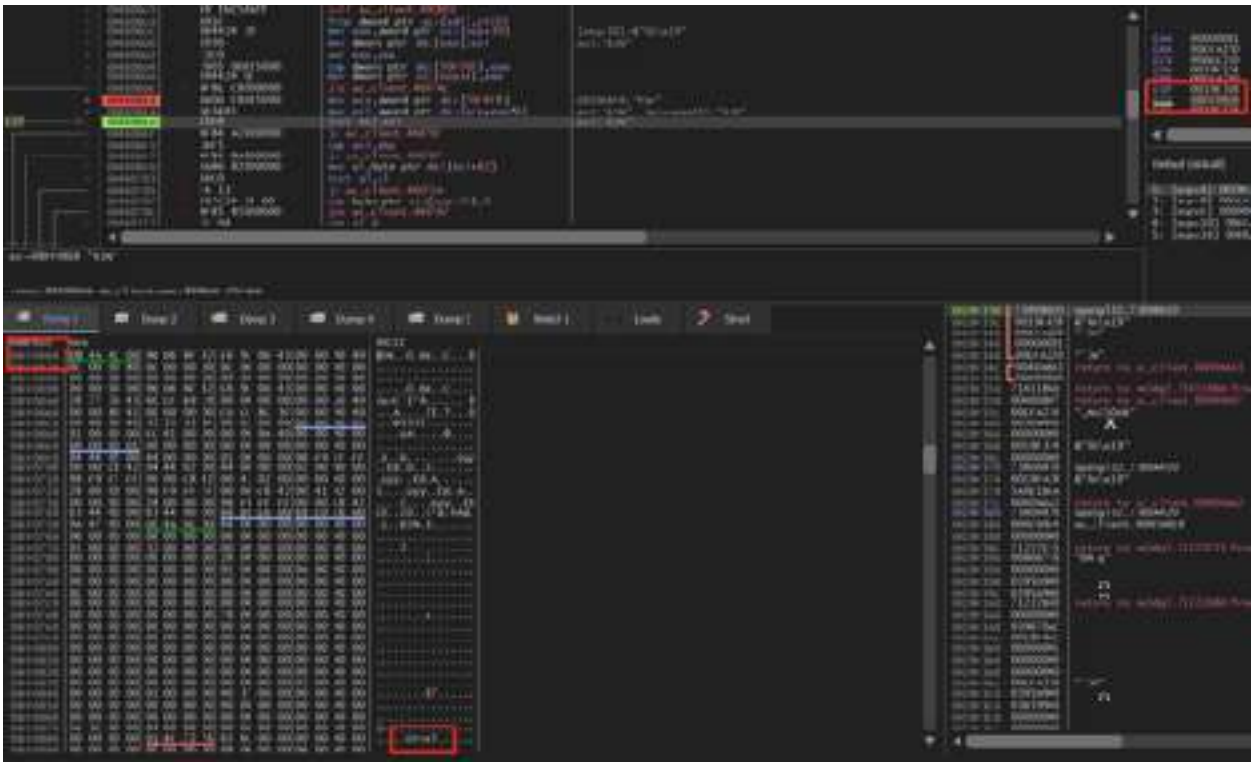
A partir disso, podemos determinar duas coisas. Primeiro, [0x50f500] terá o número atual de jogadores no jogo. Precisaremos desse valor mais tarde quando iterarmos todos os jogadores para mirar neles. A segunda é comparar o eax com esse valor. Se o eax for menor que esse valor, há uma execução jmp abaixo. Isso significa que provavelmente somos responsáveis pelo ciclo de todos os participantes ativos no código. Nas seguintes linhas, você notará o seguinte código:



Este código carrega um endereço de memória estático no ecx e, em seguida, recupera um novo endereço com base no valor do endereço combinado com o deslocamento do eax. Em seguida, carregue esse novo valor no esi. Quando esse ciclo ocorre pela primeira vez, o valor esi é 0:

Entretanto, se continuarmos e executarmos o loop novamente, o esi salvará valores diferentes:

Se verificarmos o valor esi no dump, veremos que ele está sempre perto do endereço onde o nome do jogador é salvo:

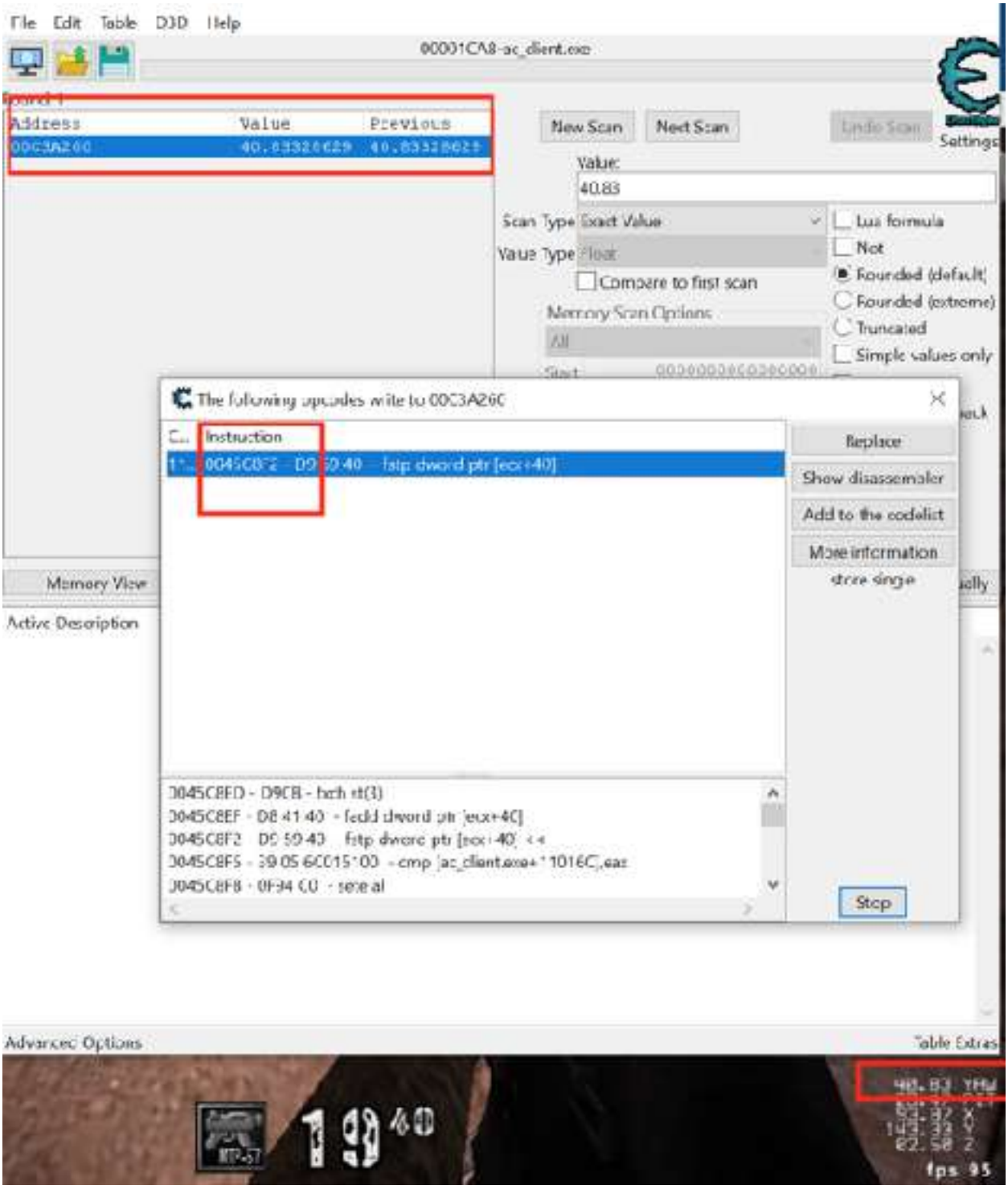


Esta é provavelmente a estrutura do jogador inimigo na memória, já que um dos valores armazenados nessa estrutura é o nome do jogador. Atualmente, confirmamos que a lista de inimigos está salva em [0x50f4f8], e cada inimigo está em [[0x50f4f8]+4, 8, C...]. Assim que encontrarmos a estrutura do jogador, inverteremos completamente o layout da estrutura do jogador na memória.

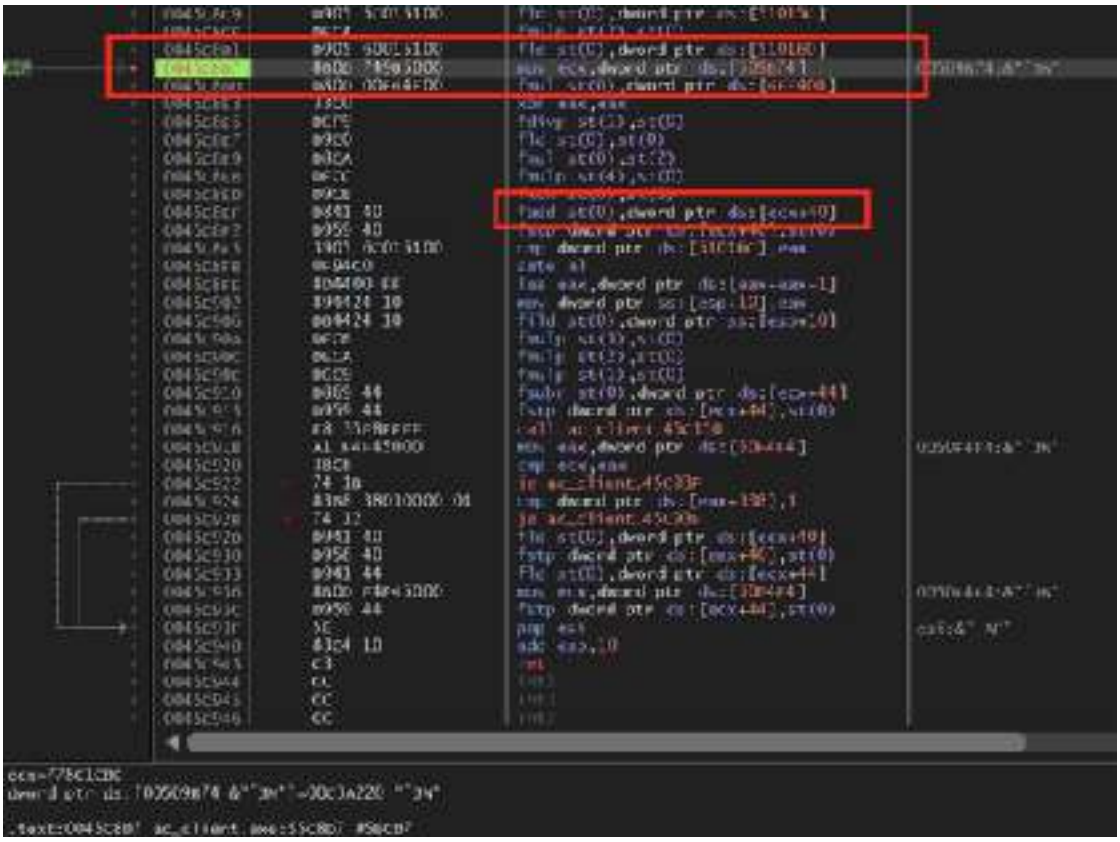
5.6.5 Localize nosso player

Em seguida, precisamos localizar o local de armazenamento do nosso próprio player na memória. Como nunca podemos nos olhar para nós mesmos, precisamos encontrar nossos jogadores de maneiras diferentes. Muitos jogos têm maneiras de imprimir sua localização e perspectiva atuais na tela. Em um cubo de ataque, isso pode ser feito com dbgpos 1. Quando ativado com showstats 1, a saída é a seguinte:

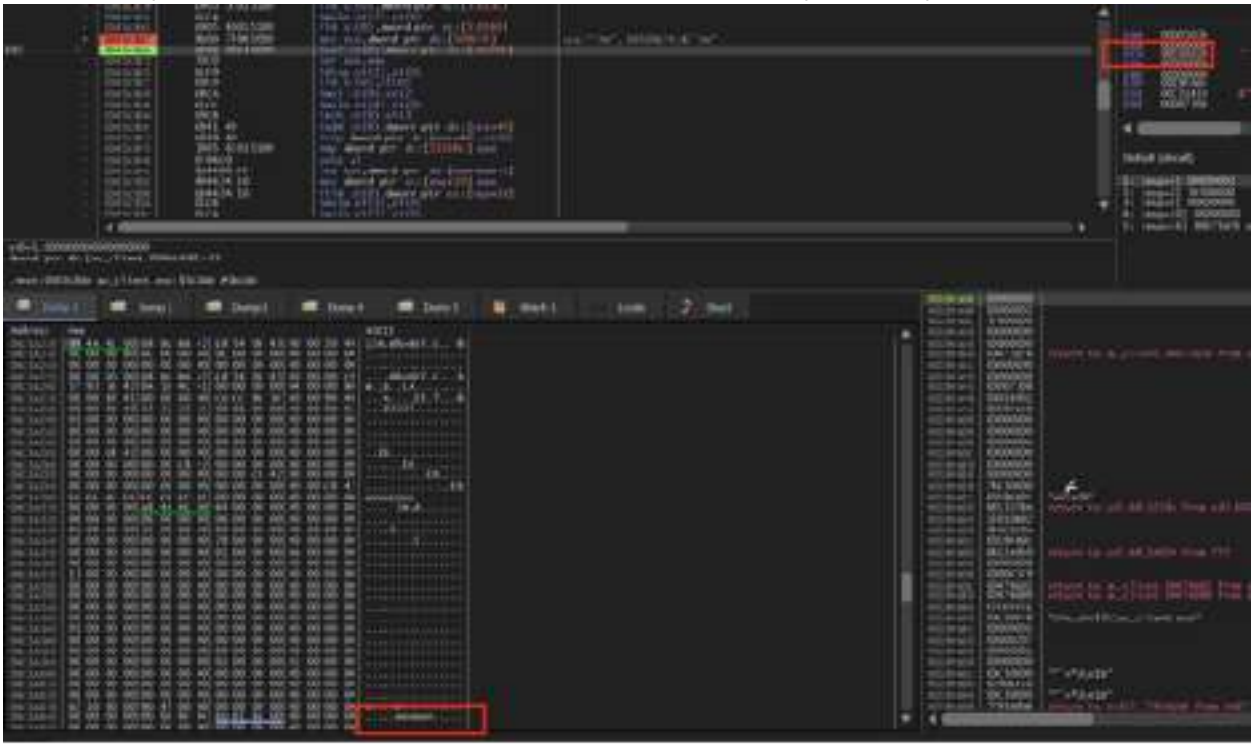
Se seu alvo não tiver esse recurso, você terá que pesquisar valores desconhecidos, filtrá-los e mover cuidadosamente seu mouse ou jogador em uma direção. Em um cubo de assalto, parece que nosso ângulo de visão é expresso por guia (esquerda e direita) e inclinação (para cima e para baixo), ambos os valores em graus. Precisamos lembrar deste modelo mais tarde. Atualmente, sabemos que nossa estrutura do jogador deve incluir esses valores. Em um mecanismo de fraude, podemos pesquisar nosso guia na memória e ver o que acessou o endereço:



A instrução fstp copia o número de ponto flutuante para o endereço especificado. Neste caso, o endereço é baseado em ecx. Se verificarmos esta instrução em x64dbg e olharmos as linhas acima, podemos encontrar a localização das configurações do ecx:



Se verificarmos a memória apontada por [0x509b74], veremos uma
O inimigo que observamos, os nomes dos nossos jogadores aparecem perto deste endereço:



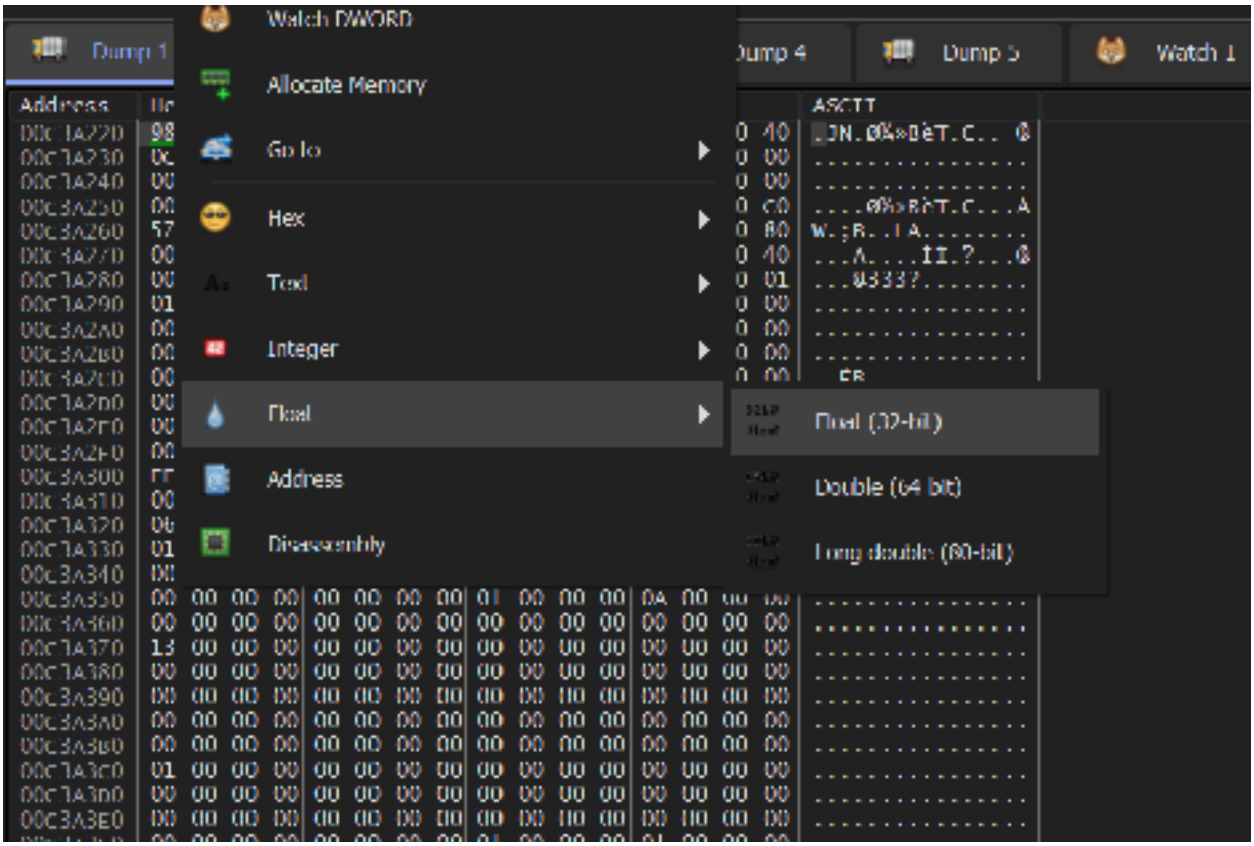
Isso provavelmente representa a estrutura do nosso jogador. Como temos mais controle sobre os valores nessa estrutura, podemos começar a revertê-los.

5.6.6 Estrutura do player de reversão

Sabemos que os jogos devem armazenar os dados de cada jogador na memória. Esses dados geralmente estão em uma parte contínua da memória. Em C ou C++, isso será expresso como uma estrutura ou classe. Por exemplo, um jogo pode definir a seguinte estrutura do jogador:

```
Player estrutural{
Ponto flutuante x;
y flutuante;
z flutuante;
guia flutuante;
espaçamento flutuante;
Modelo_texture_Path[128]; Nome do
caractere[128];
Bull está vivo;}
```

Quando visualizado em x64dbg, a estrutura aparece como um longo segmento de memória porque os dados não têm o conceito de seu tipo. Para identificar esses dados, precisamos reverter a estrutura. x64dbg permite que você modifique a representação de dados no dump. A visão padrão é uma visão hexadecimal representada por ASCII. Primeiro tentaremos encontrar o valor da nossa posição, que é representado por três valores de ponto flutuante. Podemos clicar com o botão direito e selecionar Float para exibir os dados de dumping neste formato:



Ao fazer isso, vários valores devem aparecer imediatamente, representando nossos X , Y e Z:

Da mesma forma, nossos giros e inclinações são facilmente observados:

5.6.7 Mudando nossa perspectiva

Neste ponto, já temos todos os deslocamentos necessários para começar a criar o aimbot. O primeiro passo é criar um DLL que gira continuamente nosso player. Queremos começar por aqui para garantir que posicionamos corretamente nossos jogadores e revertemos a estrutura dos jogadores corretamente. Como fizemos antes, começaremos criando uma thread que irá executá no processo do jogo:

```
#include<Windows.h>

void injectid_THREAD(){
	while(true){
//Código aimbot
		Sono (1);
	}
}

BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

	Se(fdwReason==DL_Process_Attach){
		Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL
, 0, NULL);
	}

	Retornar a
verdade;}
```

Discutimos esse código muitas vezes em capítulos anteriores. Em seguida, precisamos definir a estrutura do player na função injected_thread:

```
Player de
estrutura {}
```

Em nosso trabalho anterior, sabemos que nossos membros X, Y e Z estão localizados em base +4, base+8 e base+C, respectivamente. Não sabemos o que representam os primeiros quatro bytes, mas felizmente não precisamos saber. Em vez disso, podemos criar membros de posicionamento para um array de caracteres. Escolhemos caracteres porque eles têm 1 byte de comprimento. Podemos então criar membros de ponto flutuante para valores X, Y e Z:

```
Player estrutural{
       Caráter desconhecido 1
       [4]; Ponto flutuante x;
       y flutuante;    z
       flutuante;
}
```

Em seguida, precisamos adicionar membros de giro e pitch. Se olharmos para a memória, veremos que Z termina em 0xc3a230 e o Jyaw começa em 0xc3a260. Assim como o placeholder acima, usaremos o membro char para adicionar um preenchimento de 0x30 bytes antes de adicionar vias e pitch:

```
Player estrutural{
Caráter desconhecido 1[4];
       Ponto flutuante x;
       y flutuante;
       z flutuante;
Caráter desconhecido 2[0x30
       ]; guia flutuante;
       Espaciamento
flutuante;}
```

Em seguida, criaremos um ponteiro desta estrutura, que usaremos para mapear a memória do jogo para a parte posterior:

```
Player * Player = vazio
```

À medida que nossa estrutura é criada, agora podemos mapear a memória do jogo do jogador em nossa estrutura. Primeiro, criaremos um ponteiro para 0x509b74 no loop while, porque isso representa o endereço do player:

```
DWORD*player_offset=(DWORD*)(0x509B74);
```

Em seguida, desreferiremos este ponteiro para obter o valor do endereço do player. Em seguida, mapeamos o endereço da referência para o nosso ponteiro de estrutura do player. Isso armazenará os valores que observamos no dump nesta estrutura para que possamos referi-los em nosso código.

```
player*(player*)(*player_offset);
```

Por fim, adicionaremos membros de guia a um loop para que nosso player gire em um círculo:

```
Player->Jame++;
```

A partir daqui podemos construir e injetar este DLL. Nosso player agora gira em círculo , indicando que invertimos corretamente a estrutura do player.

Amirar em torno de 5,6,8

Com tudo isso, podemos criar a primeira versão do aimbot. Esta versão mira em um oponente para a esquerda e para a direita. Ao testar isso, certifique-se de criar um jogo para dois jogadores com você e um robô. Isso lhe dará certeza sobre matemática.

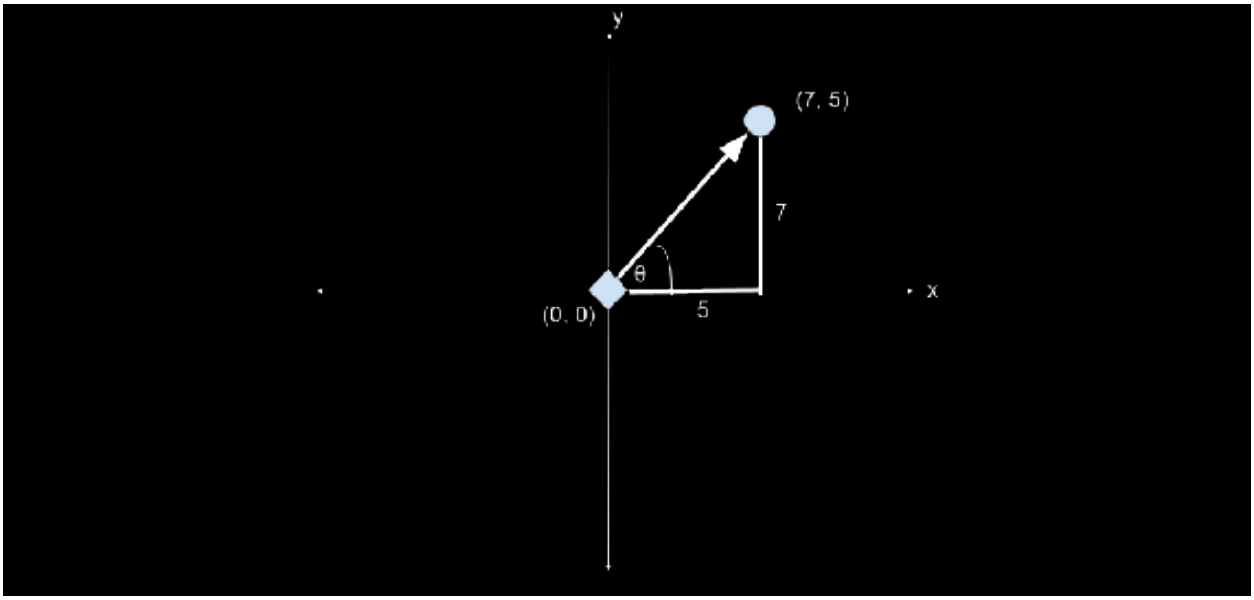
Primeiro, mapearemos o primeiro inimigo na estrutura do jogador usando o mesmo método que antes. Ao reverter o código, determinamos que o primeiro inimigo está em +4:

```
DWORD*enemy_list=(DWORD*)(0x50F4F8); DWORD*enemy
_offset=(DWORD*)(*enemy_list+4); Player*enemy=(
Player*)(*enemy_offset);
```

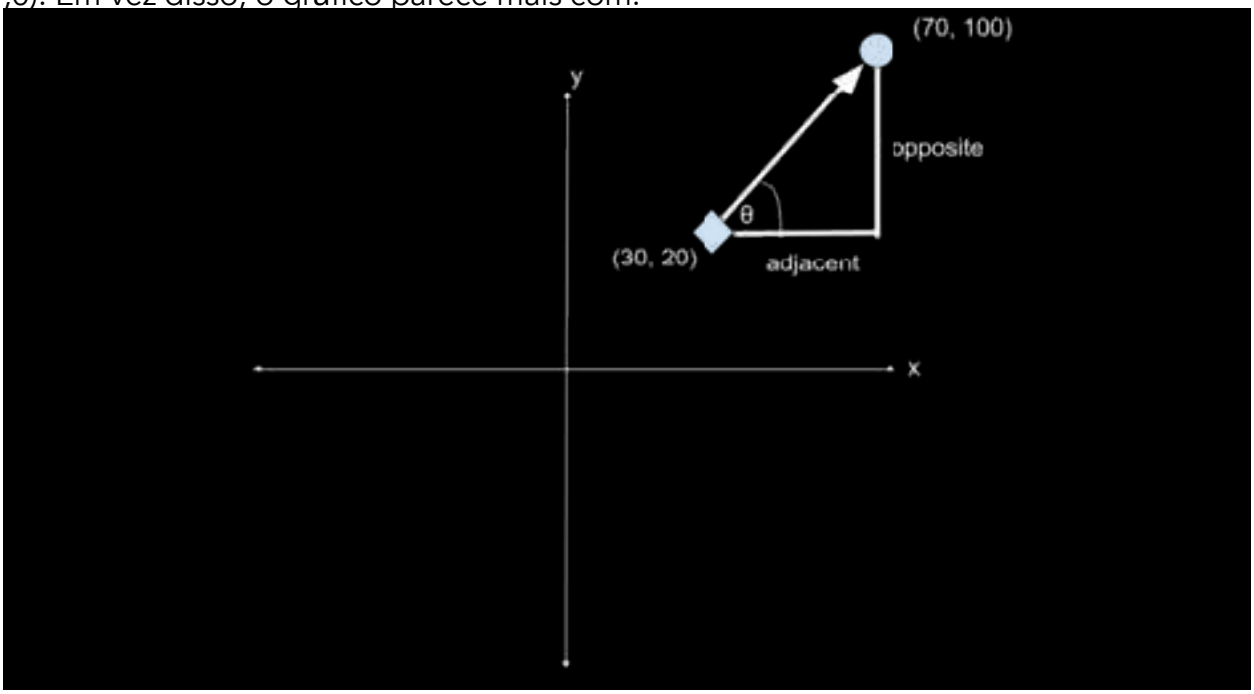
Um problema com nossos códigos anteriores era que se não estivessem no jogo, podíamos travar. Isso ocorre porque estamos acessando memória inválida. Para evitar isso, verificaremos antes de continuar para garantir que ambos os nossos ponteiros sejam válidos:

```
Se(player!=NULL!=NULL){
```

No início deste capítulo, temos a seguinte figura:



Nesta imagem, sabemos as distâncias relativas e adjacentes com base na posição do inimigo. Entretanto, como vemos durante a reversão, nossa posição nunca é (0 ,0). Em vez disso, o gráfico parece mais com:



Se tentarmos explorar posições inimigas, nossos cálculos serão incorretos. Em vez disso, precisamos determinar esses valores subtraindo a posição do inimigo da posição do jogador. Isso nos dará alguns valores, como se o jogador estivesse sempre em (0, 0), ou na posição absoluta entre nossos jogadores e inimigos (abspos):

```
float abspos_x=inimigo ->x-player->x;
float abspos_y=inimigo->y-player->y;
```

Em seguida, podemos usar a função atan2f para calcular o tangente. Usamos essa função em vez de atanf porque ela lida com casos em que abspos_y é menor que 0. Como o antitangente é uma operação sem sinal (ou seja, não tem o conceito de positivo ou negativo), se o inimigo estiver bem atrás de nós, nosso aimbot mira na direção oposta. Podemos verificar isso manualmente verificando abspos_y, mas atan2f processará esse cálculo para nós:

```
#Incluindo <math.h>

...
Ángulo de azimute flutuante xy=atan2f (abpos_y, abpos_x);
```

A função atan2f gera um valor de radiandade. Ao dar ré, vemos que o jogo representa nossa guinada como um grau. Para converter o radianismo em um valor de grau, podemos multiplicar o radianismo por (180/):

```
#Definição M_PI 3.14159265358973323846

...
Cruzamento flutuante = (flutuante) (ângulo azimute xy*(180,0/M_PI));
```

Por fim, podemos definir a curva do player para este valor:

```
Jogador -> Juramento = Juramento;
```

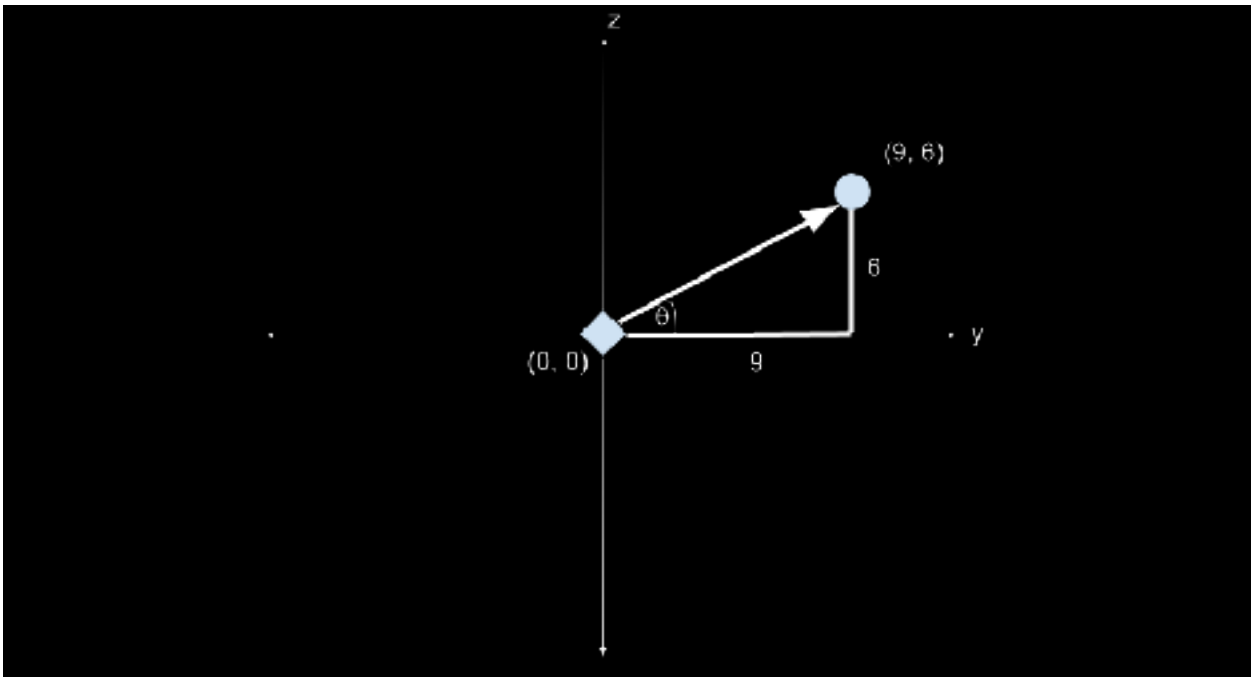
Se você injetar esse código no jogo, notará que seu alvo está perto do jogador, mas sempre mantém um número consistente de pixels para a esquerda ou para a direita, dependendo de onde você está. Isso ocorre porque, em nosso modelo gráfico, assumimos que 0 esteja voltado para a frente. No entanto, se você entrar em um jogo sem hackers, notará que o giro inicial do seu jogador é 90. Para compensar isso, podemos simplesmente adicionar 90 à curva calculada:

```
Jogador ->Giro = Giro + 90;
```

Com essa mudança, podemos correr pelo mapa e continuar fixando um jogador. No entanto, se saltarmos para cima ou para baixo ou subiremos uma ladeira, miraremos acima ou abaixo do inimigo . Nosso próximo passo é definir corretamente nossos valores de tom (ou para cima e para baixo).

5.6.9 mira para cima e para baixo

Quando abordamos esse problema pela primeira vez, rapidamente definimos a perspectiva para cima para baixo para eliminar o eixo Z. Para calcular nossos ângulos de cima para baixo, agora fixaremos nossa perspectiva da direita para a esquerda (ou seja, do lado direito do jogador, olhando diretamente para o lado direito do jogador). No novo gráfico abaixo, o mais importante a prestar atenção são os diferentes valores de eixo:



Podemos usar métodos semelhantes aos ângulos esquerdo e direito para calcular os ângulos superior e inferior Primeiro, teremos distância absoluta:

```
float abspos_z=inimigo->Z-jogador->z;
```

Então, como antes, calcularemos a tangente inversa. Ao contrário da curva, nossa inclinação inicial começa em 0, então não precisamos adicionar nenhum valor a ela:

```
Azimuto flutuante _z=atan2f (abspos_z, abspos_y);
player->pitch=(float) (azimuth_z*(180.0/M_PI));
```

Se você injetar esse código, ele parece funcionar inicialmente. Entretanto, ao se aproximar de um inimigo, seu jogador de repente olhará diretamente para ele ou o olhará diretamente . Isso ocorre porque o valor Z do jogo é muito limitado. Por exemplo, a maioria dos mapas do jogo tem valores Z entre 0 e 6. Quando o valor de Y se torna muito pequeno, a equação resultante eventualmente se inclinará. Imagine uma situação em que a diferença entre os valores Z é 3, mas a diferença entre os valores Y é 1 ou arctan (3/1). Isso é decomposto em 75°, o que na verdade é direto para cima no ar quando se trata de inclinação.

Para explicar esse comportamento, analisaremos o valor de Y e garantiremos que ele seja bastante grande. Caso contrário, usaremos X. Isso não é perfeito, mas ajudará a aliviar alguns problemas. Também garantiremos que o valor seja positivo, independentemente de:

```
Se (abpos_y<0){abpos_y*=
-1;
}
Se (abspos_y<5){se (
abspos_x<0){abspos_x* =
-1;}

abpos_y=abpos_x;}
```

Agora você notará que pode correr diretamente em direção ao inimigo, e seu alvo não salta no ar. Nosso robô alvo agora trabalha apenas para um inimigo.

5.6.10 mais de inimigos

Com essa base, podemos modificar nosso aimbot para lidar com vários inimigos. Para isso, mudaremos o código para iterar a lista de inimigos, selecionaremos um inimigo para mirar e definiremos nossos giros e inclinações para mirar neles. Ao escolher inimigos, sempre escolheremos o inimigo mais próximo de nós. Este nem sempre é o melhor cenário. Por exemplo, se um inimigo estiver no salão e outro inimigo estiver atrás da parede ao lado de nós, nosso aimbot sempre escolherá o inimigo atrás da parede. Entretanto, para este capítulo, essa abordagem é a mais fácil de implementar.

Para encontrar o inimigo mais próximo de nós, calcularemos a distância euclídia entre nossos jogadores e inimigos. Quanto menor o valor, mais próximo o inimigo está de nós:

```
  float Euclidea_Dostance(float x, float y){
        return sqrtf((x*x)+(y*y));
}
```

Como precisamos iterar a lista de inimigos, criaremos uma variável para salvar a distância do inimigo mais próxima, bem como seus valores de curvas e pitch associados:

```
while(true){
DWORD*player_offset=(DWORD*) (0x509B74); player
* (player*)(*player_offset);
...
float closest_player=-1.0f; float
closest_yaw=0,0f; float closest_
pitch=0,0f;
```

No início deste capítulo, determinamos o endereço para salvar o número atual de jogadores no jogo. Agora podemos finalmente usar esse valor:

```
int*currente_Players=(int*)(0x50F500);
```

Agora podemos iterar todos os inimigos do jogo. Ao contrário de antes, sempre adicionamos 4, mas agora multiplicamos o índice de loop atual adicionado por 4, o mesmo que o jogo faz:

```
for(int i=0;i i<*currente_players;i++){
DWORD*enemy_list=(DWORD*) (0x50F4F8);
DWORD*enemy_offset=(DWORD*) (*enemy_list+(i*4)); Player*
enemy=(Player*)(*enemy_offset);
```

Então podemos calcular a posição absoluta como antes. No entanto, antes de calcular o guinho ou a inclinação, calculamos a distância do nosso jogador até o inimigo e garantimos que ele seja o inimigo mais próximo. Se sim, definiremos closest_player valem a distância até eles para inspeção futura:

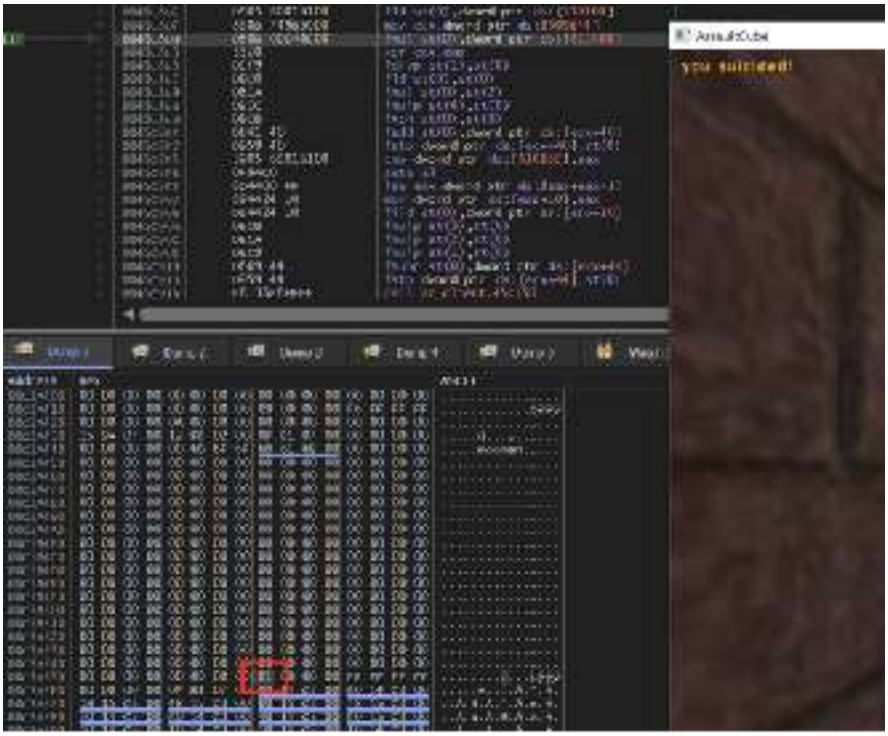
```
Temperatura float = Euclide_distance (abspos_x, abspos_y); Se (
closest_player==-1.0f||temp_diaser<closest_player){closest_
player=temp_diaser;
```

Em seguida, os armazenaremos em variáveis em vez de definir diretamente os curvas e inclinações do jogador. Depois que iteramos todos os inimigos, definimos o giro e a inclinação do jogador. Isso garante que não continuemos piscando entre vários inimigos:

```
closest_yaw=yaw+90;
...

closest_pitch=(float)(azimuth_z*(180.0/M_PI));
...

Player -> Jame = Jame mais
próximo; player->pitch=closest
_pitch; Sono (1);
```

Agora temos um aimbot em funcionamento que iterará por vários inimigos e mirará corretamente no mais próximo nos eixos X e Y.

Por fim, podemos adicionar uma verificação para verificar se o inimigo ainda está vivo, garantindo que mudamos do alvo imediatamente ao atirá nele. Esse valor pode ser encontrado observando a estrutura do jogador para encontrar o valor que muda quando você estiver vivo ou morto. Depois de cometer suicídio várias vezes, você notará que há um valor definido para 0 quando você estiver vivo e para 1 quando você estiver morto:



Podemos adicioná-lo à nossa estrutura do player para garantir que o cancelamos corretamente:

```
guia flutuante;
      espaçamento flutuante;
      char unknown 3[0x2f0];
      int está morto;
};
```

Podemos então verificar esse valor na verificação inicial para garantir que o jogador e o inimigo sejam válidos:

```
Se (jogador! = vazio & inimigo! =vazio & ! Inimigo ->morre){
```

Códigos completos podem ser comparados no Apêndice A.

5.7 Sem recuo

5.7.1 Objetivos

Nosso objetivo neste capítulo é atacar o Cubo Mágico 1,2.0,2.

5.7.2 Identificação

Nosso objetivo neste capítulo é criar um hacker sem recuo, um hacker que elimina recuo ao disparar. O recuo é definido como o movimento automático do campo de visão do jogador para cima quando uma arma é disparada.

5.7.3 Compreensão

Ao disparar armas em FPS, a maioria dos jogos usa efeitos diferentes:

- Recuperação (movimento para cima e para baixo)
- Difusão (linha cruzada alargada, lentes distribuídas aleatoriamente)
- Empurre para trás (o jogador é empurrado para a direção oposta ao seu tiro)

Nosso foco neste capítulo é eliminar o recuo.

Na maioria dos jogos, esses efeitos são aplicados continuamente após cada tiro. O recuo geralmente aumenta o ângulo de visão superior e inferior do jogador aumentando um certo valor. Como o ângulo de visão geralmente é um número de ponto flutuante, essa operação geralmente adota a seguinte combinação:

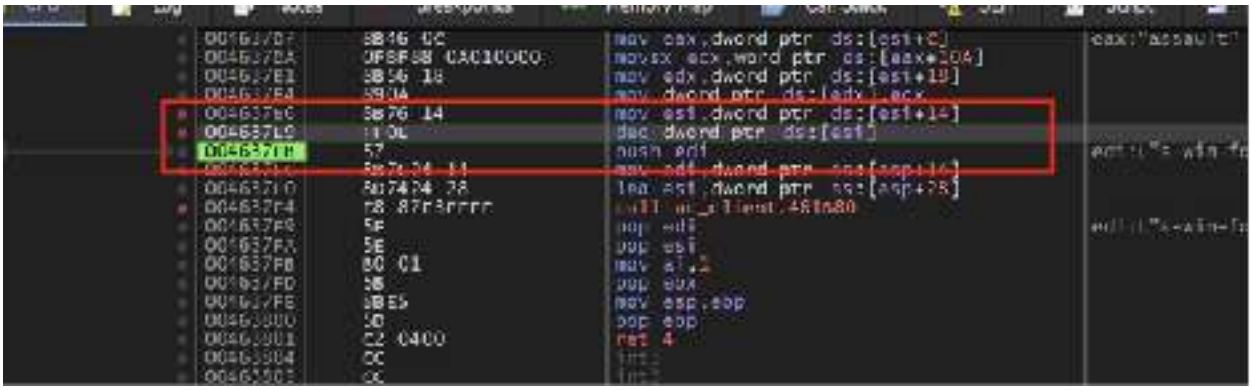
```
Recuperação fld; Carregue a quantidade de reversão em st(0) fadd st(0), players_y_
view_angle; Adicione recoil_amount ao ângulo de visão fstp players_y_view_angle,
st(0); Armazene os resultados na perspectiva
```

Ao contrário dos números inteiros, os valores de ponto flutuante devem ser empurrados para uma pilha de registro especial antes de serem operados, chamada pilha FPU. Entretanto, como nas instruções normais, o recuo não será aplicado ao jogador se esse código for excluído.

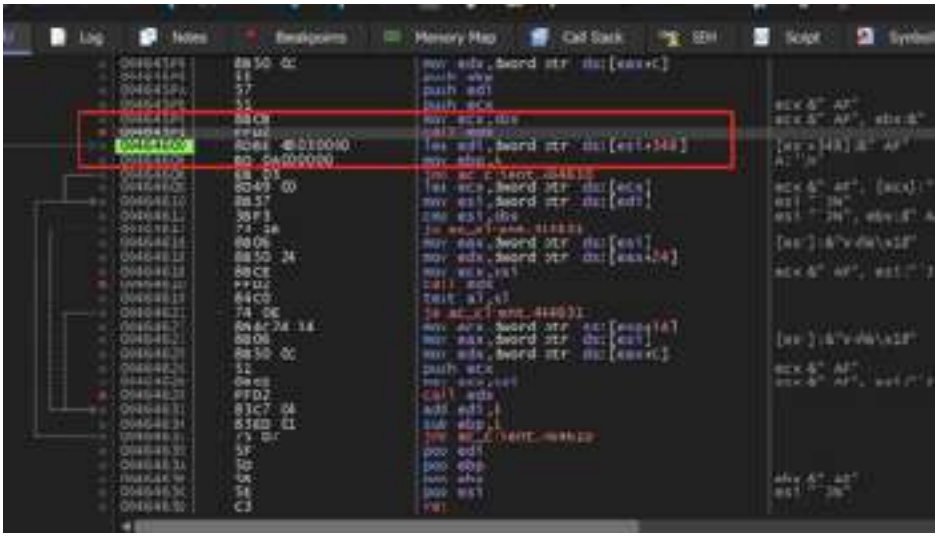
Ao disparar armas, o jogo executa várias funções, incluindo reproduzir sons, exibir animações de tiro e reduzir a munição do jogador. Esses recursos geralmente estão localizados perto de recursos que aplicam recuo à visão do jogador. Podemos usar essas funções para ajudar a localizar códigos de rebote. Temos várias maneiras de localizar esse código. Neste capítulo, usaremos o código responsável por reduzir a munição do jogador porque esse valor é fácil de pesquisar.

5.7.4 Função de posicionamento e disparo

Comece um jogo de Cubo Mágico usando um mecanismo de trapaça para localizar sua quantidade atual de munição, usando o mesmo método discutido no Capítulo 1.5. Assim que isso for reconhecido, o x64dbg é anexado ao ASSULT Cube e o ponto de interrupção do hardware para gravação é definido no endereço reconhecido. Ao retornar ao cubo de assalto e abrir fogo, o ponto de interrupção deve ser lançado nas seguintes posições:

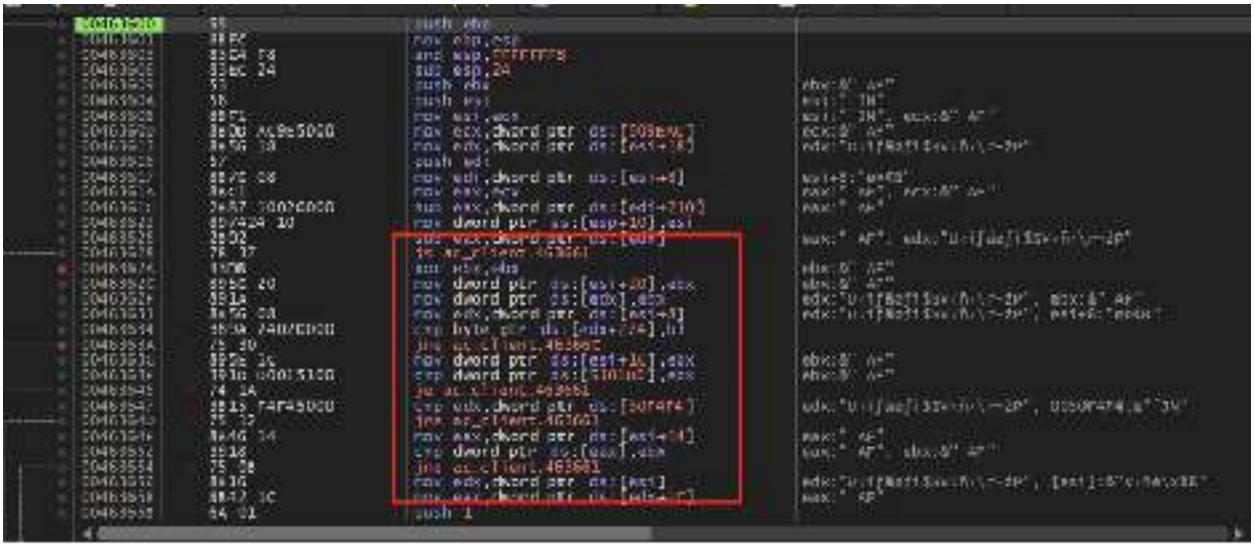


Podemos ver que esse código é o motivo da redução da quantidade de munição. Se saírem deste código usando o Excyte until Retour/step, veremos que a localização da chamada está aqui:

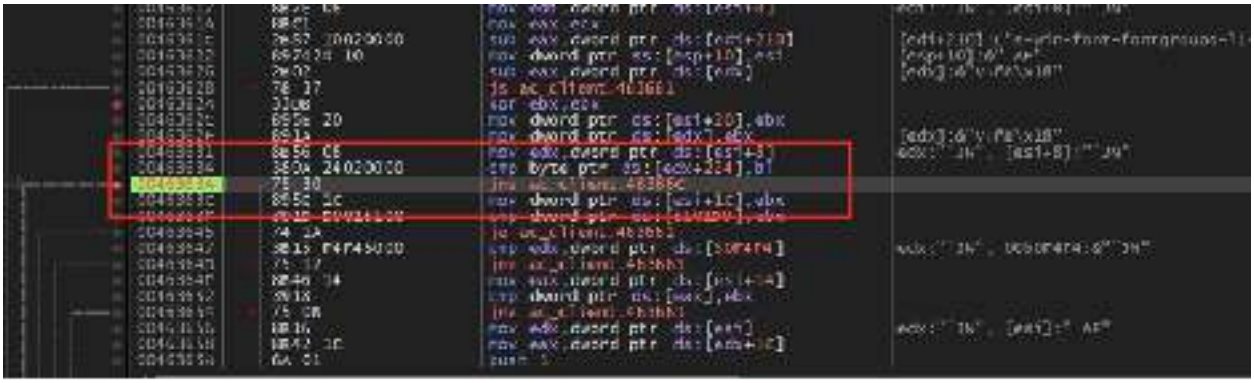


Em seguida, vamos determinar o contexto no código. Queremos determinar se estamos no código que é responsável apenas por definir a quantidade de munição ou no código geral de tiro. Podemos fazer isso definindo um ponto de interrupção na instrução de chamada edx. Depois disso, podemos ver o código sendo chamado constantemente, independentemente de estarmos disparando ou não. Isso significa que estamos muito avançados e precisamos explorar esse recurso.

Se entrarmos na chamada em uma única etapa após o acionamento do ponto de interrupção, podemos ver que a função tem vários ramos:

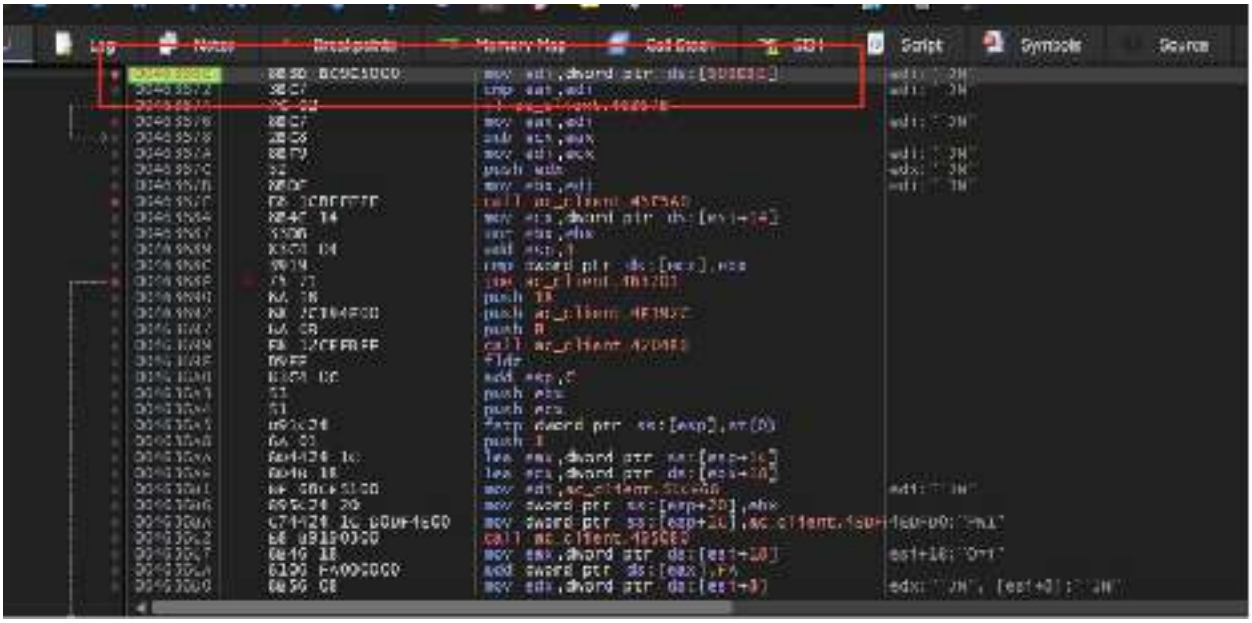


Se executarmos o código em uma única etapa, podemos ver que se não acionarmos, o jmp em 0x4633A não será ocupado:



Se mudarmos para jmp e retornarmos ao jogo, notaremos que nossos jogadores agora Continue atirando, mesmo que pressionemos e seguremos o botão do mouse. Este jmp parece ser

Responsável por verificar se os jogadores abriram fogo. Se seguirmos este jmp, podemos ver que ele pulará uma declaração de return e saltará para a seguinte instrução:



Quando definimos um ponto de interrupção nesta instrução, vemos que esse código só é chamado quando o atiramos ativamente.

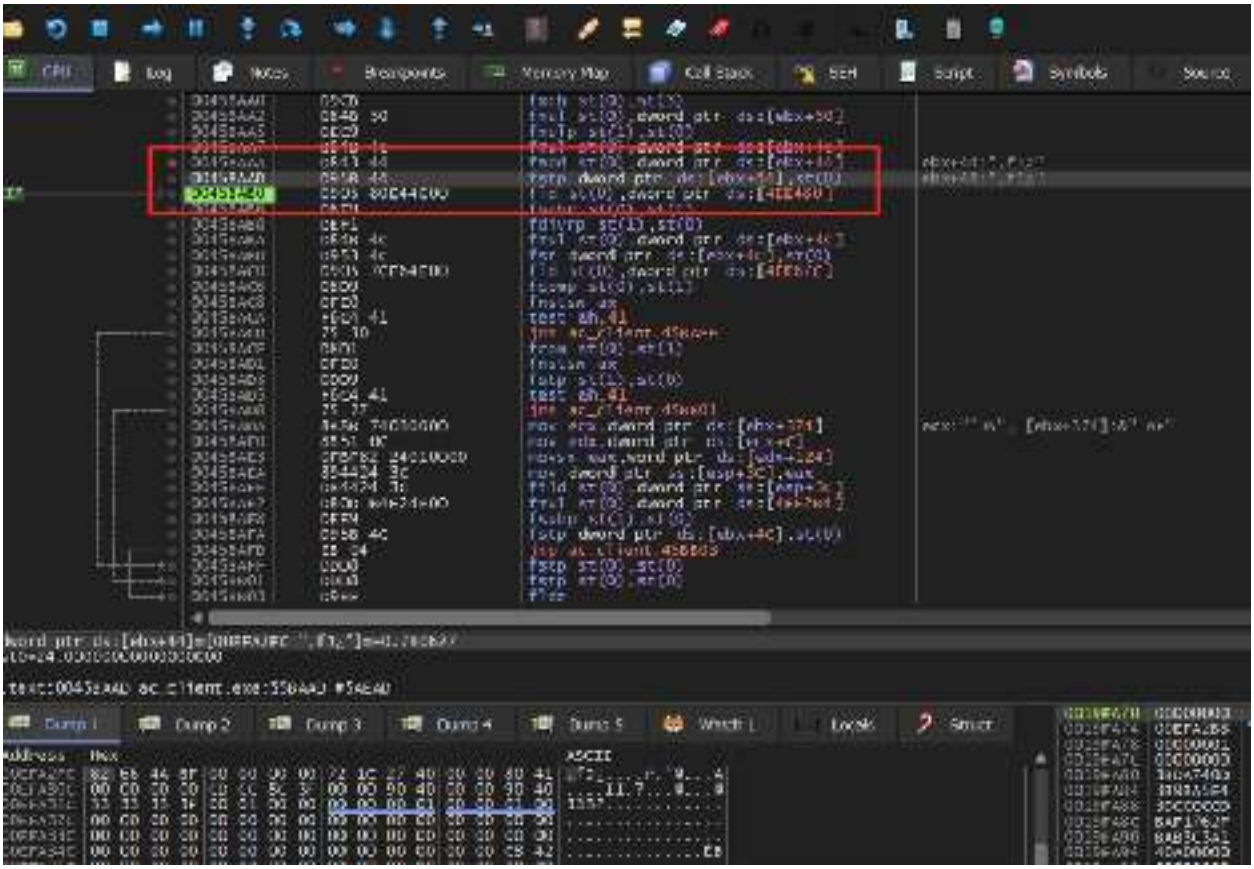
5.7.5 Posicionamento e retrocesso

Agora encontramos o início do código de lançamento da arma. Também podemos ver que o último comando no código de tiro da arma é responsável por reduzir a munição. Portanto, em algum lugar entre essas duas instruções está o código responsável por aumentar o recuo.

Embora possamos percorrer esse código passo a passo para identificar instruções, podemos usar métodos mais rápidos. Sabemos que o comando de contra-ataque deve modificar o valor de guinada do jogador. Depois de atingir o ponto de interrupção do lançamento da arma, se definirmos um ponto de interrupção no valor do guinho, podemos continuar executando e esperar que o ponto de interrupção apareça. Isso nos impede de executar grandes quantidades de código em uma única etapa.

É importante ressaltar que definimos pontos de interrupção no valor de guia somente depois que o código de gatilho for ativado. Assim Cube, como muitos outros jogos, continua gravando valores de guia. Se simplesmente definirmos um ponto de interrupção nele e não estivermos no código de gatilho, terminaremos em outro código.

Podemos usar o mesmo método discutido no capítulo anterior ou localizar o endereço do valor de guia pesquisando no mecanismo de trapaça. Depois disso, defina um ponto de interrupção 0x46366C no início do código de gatilho. Então, dispare um tiro para liberar o ponto de interrupção. Quando isso acontecer, defina um ponto de interrupção de gravação no endereço onde o valor do guia está localizado . Continue executando, o ponto de interrupção de gravação deve aparecer nos seguintes códigos:

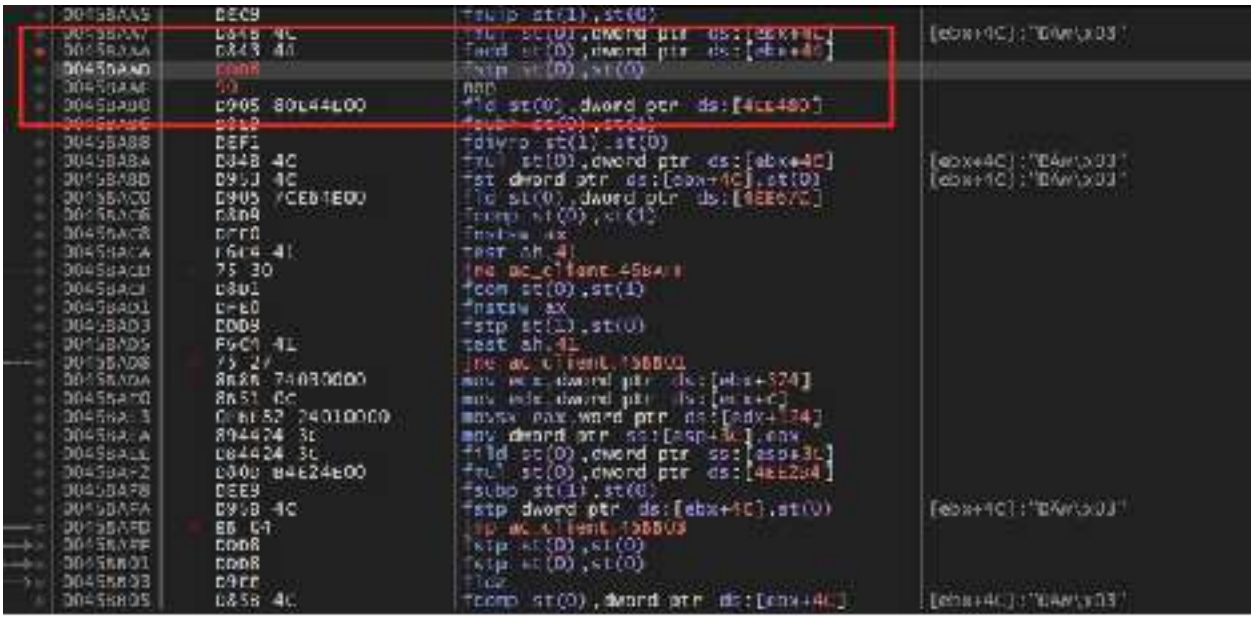


Podemos ver que esse código corresponde ao padrão que esperávamos. Neste código específico, dword ptr ds: [ebx+0x44] é responsável por salvar o valor de jaula do player. O valor de reversão permanece no topo da pilha FPU, com st0 apontando para essa pilha.

A operação para calcular o recuo parece consistir em várias instruções. Embora possamos investigar exatamente como o recuo é definido, podemos pular esse processo para lançar um ataque de hacker sem recuo e simplesmente impedir que o recuo seja colocado no valor de guinada do jogador.

A instrução fstp é responsável por exibir o valor superior da pilha FPU no endereço fornecido. Como não queremos destruir a pilha, não queremos a instrução nop porque haverá um valor extra na pilha. Pelo contrário, nós

O valor no topo da pilha é pop-up para st0. Como st0 é posteriormente definido na próxima instrução, isso fará com que o valor desapareça de fato:



Com essa mudança, você notará que não terá mais recuo no jogo.

5.7.6 Alteração de recuo

Por fim, podemos escrever um DLL para fazer essa mudança automaticamente. Como esse hacker só exige que alteremos bytes em uma instrução, podemos usar o mesmo modelo que apresentamos no Capítulo 4.2. Para este hacker, alteraremos o código editado em 0x45 baad bytes para o mesmo valor que observamos em x64dbg:

```
#include<Windows.h>

unsigned char new_bytes[3]={0xDD, 0xD8, 0x90};

BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

	DWORD old_protect;
	unsigned char*Hook_Localização=(unsigned char*)0x45bad;

	Se(fdwReason==DL_Process_Attach){
		Projeto Virtualítico ((void*)Hook_localizador, 3 página_Express_
Readwrate,&old_Projeto);
		for(int i=0; i<sizeof(new_bytes);i++){
```

```
                    *(hook_localização+i)=new_bytes[i];
                }
        }

        Retornar a verdade;
}
```

O código deste hacker também pode ser encontrado no Apêndice a.

5.8 Hacker de Radar

5.8.1 Objetivo

Nosso objetivo neste capítulo é atacar o Cubo Mágico 1,2.0,2.

5.8.2 Identificação

Nosso objetivo neste capítulo é criar um hacker de radar, um hacker que exibe inimigos e jogadores amigáveis no radar.

5.8.3 Compreensão

Muitos jogos FPS têm um radar que exibe um ícone para cada jogador em uma versão menor do mapa atual. Ao jogar com um time, esses radares só exibem jogadores que estão no mesmo time que jogadores ativos. No Capítulo 5.6, descobrimos a localização da lista de jogadores no jogo. Ao desenhar ícones de jogador no radar, a maioria dos códigos do jogo iterará esta lista de maneiras semelhantes aos seguintes blocos:

```
void draw_radar(){
for(int i=0;i <max_players;i++){
if(player_list[i]->team===current_player->team){//
Desenhado no radar
}}}
```

Se encontrarmos esse código, podemos alterar a condição if, independentemente do time , sempre desenhando jogadores no radar. Para encontrar o código, primeiro precisamos identificar os membros da equipe na estrutura do jogador. Podemos então definir um ponto de interrupção no acesso para identificar onde o membro é acessado no código.

5.8.4 Localize a equipe do jogador

Podemos usar duas maneiras de localizar a equipe do jogador dentro da estrutura do jogador:

1. Use o mecanismo de fraude e pesquise por valores iniciais desconhecidos. Em seguida, alterne-se entre equipes para pesquisar valores alterados.
2. Use x64dbg e localize a estrutura do player no dump. Então, alterne-se entre as equipes e procure um membro que mude.

Já introduzimos esses dois métodos em capítulos anteriores. Usando as técnicas discutidas anteriormente, você deve ser capaz de identificar membros alternando entre 0 e 1 com base na sua equipe. Este membro é relativamente próximo do membro que determinamos anteriormente para saber se o jogador está vivo.

5.8.5 Função de radar de posicionamento

Sabemos que a capacidade de radar deve acessar a equipe do jogador. Portanto, podemos colocar um ponto de interrupção nas visitas dos membros da equipe que acabaram de ser confirmadas. O ponto de interrupção deve aparecer imediatamente. No entanto, se você continuar executando várias vezes, verá pontos de interrupção aparecendo em partes completamente diferentes do código. Isso provavelmente ocorre porque várias partes do código acessam o membro. Nosso próximo passo é identificar a parte do código responsável pelo desenho do radar.

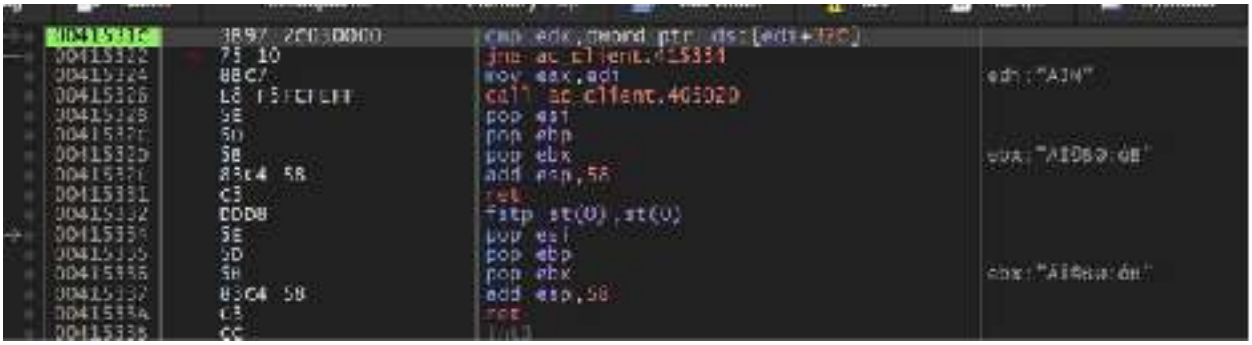
Com base no pseudocódigo que descrevemos acima, podemos fazer algumas suposições sobre o código que estamos procurando:

1. Ele terá instruções cmp (ou test) seguidas por saltos condicionais (je, jne, jg, etc.).

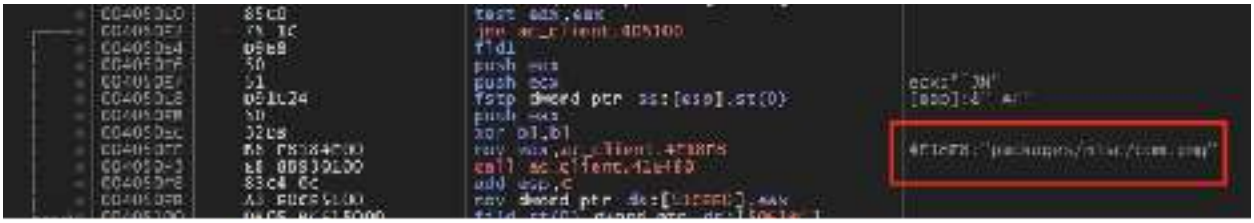
2. Como envolve muitas operações, ele chama uma função ou tem uma quantidade considerável de código.
3. É mais provável que ele use operações de ponto flutuante (fld, fstp) para localizar ícones no radar.

Podemos usar esses recursos para ajudar a encontrar onde nos importamos no código. Como estamos interrompendo a execução do programa, os pontos de interrupção não aparecem de maneira consistente. Neste capítulo, verificaremos cada segmento de código na ordem que encontramos ao escrever este capítulo. No seu ambiente, a ordem dos POP provavelmente será diferente.

O primeiro pop aparece no seguinte código:

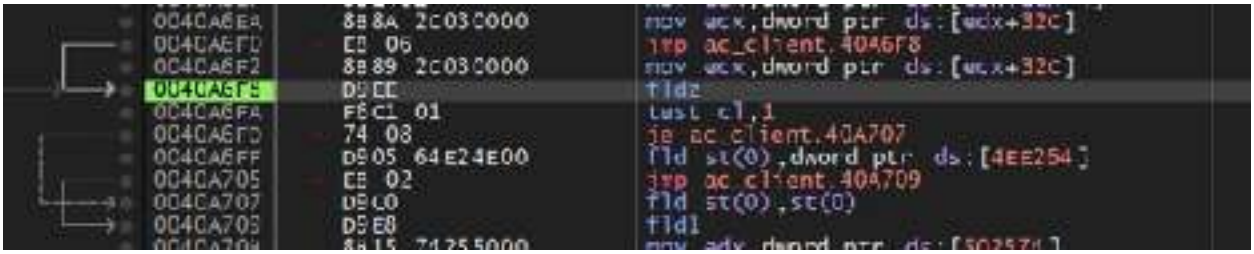


Inicialmente, isso parecia um exame de várias condições com as quais estamos preocupados . Entretanto, se notarmos o comando JNE em 0x415322, notaremos que não há alterações no jogo. Se você navegar pela chamada em 0x415326, verá o seguinte código:

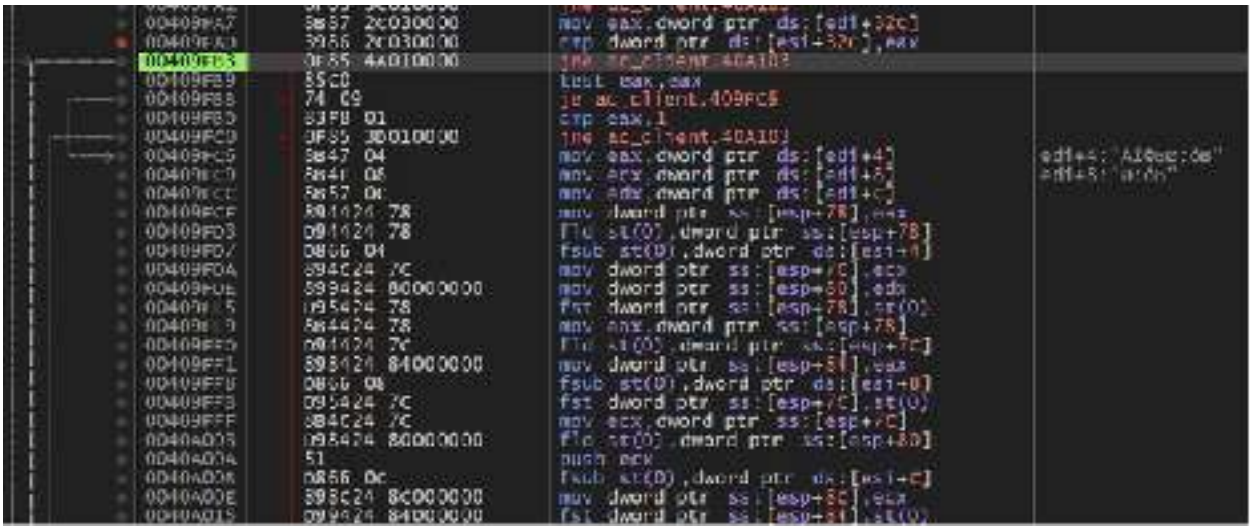


A partir dessa constante de string, podemos presumir que esse código está relacionado ao desenho de símbolos de bate-papo de voz (ou comunicação) no radar. Agora seguiremos para o próximo lugar:

Para nossos propósitos reversos diretos, testar o próprio registro é o mesmo que comparar o registro com 0. Aqui, vemos que se a equipe do jogador for definida para 0, ou equipe CLA, o código executará um ramo. As operações de desenho por radar devem ser realizadas com base em valores armazenados na estrutura do player, não em valores estáticos. Agora podemos prosseguir para o próximo lugar:



Verificando este código, vemos que ele está realizando uma operação semelhante ao código anterior. Após carregar o valor da equipe do jogador no ecx, o código compara o valor com 1 ou compara a equipe RSVF com o teste cl, comando 1. A lógica aqui é a mesma do parágrafo anterior, então vamos verificar a próxima posição:



Assim como na nossa primeira posição, este parece um candidato promissor. O comando cmp no topo compara a equipe do nosso jogador atual com a eax, que parece estar carregando o mesmo deslocamento de equipe de outra estrutura de dados (possivelmente outro jogador). Também vimos várias operações de ponto flutuante que podem ser responsáveis por colocar ícones no radar. Vamos dar uma olhada no que acontece se excluíssemos as instruções jne:

Se você voltar ao Cubo de Ataque, deve notar que agora pode ver todos os jogadores no radar, incluindo aqueles que não estão na sua equipe. Encontramos o código do radar responsável.

5.8.6 Alteração de código

Como esse ataque de hacker exige apenas que escrevamos bytes no endereço de memória, podemos usar as mesmas técnicas discutidas no Capítulo 5.7:

```
#include<Windows.h>

Caracteres sem símbolos new_bytes[3]={0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
```

```
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      DWORD old_protect;
      unsigned char*hook_localização=(unsigned char*)0x409FB3;

      Se(fdwReason==DL_Process_Attach){
            Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
            for(int i=0; i<sizeof(new_bits); i++){*(hook_
                  localização+i)=new_bits[i];
            }
      }

      Retornar a
verdade;}
```

5.9 ESP

5.9.1 Objetivos

Nosso objetivo neste capítulo é atacar o Cubo Mágico 1,2.0,2.

5.9.2 Identificação

Nosso objetivo neste capítulo é criar um hacker ESP, um hacker que exibe informações do jogador acima da cabeça do jogador. Essas informações incluem a saúde do jogador, o nome ou a arma usada atualmente e também são exibidas na parede.

5.9.3 Compreensão

No Capítulo 5.6, criamos um aimbot que funciona calculando o ângulo entre o jogador e o inimigo e, em seguida, definindo o ângulo atual do jogador como o ângulo calculado. Para jogos onde a câmera está sempre ligada ao campo de visão do jogador, como FPS, podemos usar esses mesmos ângulos para criar ESP.

Usaremos essas diferenças de ângulo para converter a posição 3D do inimigo no mundo para uma posição 2D em relação à vista do nosso jogador, em vez de definir a perspectiva do jogador. Então desenharemos o texto neste local.

A abordagem discutida neste capítulo tem várias deficiências, mas demonstra os conceitos fundamentais usados em ESP. Uma maneira mais precisa é usar o viewmatrix do jogo.

5.9.4 Porto de Visão

Aqui estão as seguintes cenas de "Cubo de Ataque":

Sabemos no capítulo anterior que o inimigo que vemos na cena acima tem uma posição 3D no mundo representada pelas coordenadas X, Y e Z. No entanto, quando jogamos o Assault Cube, o jogo precisa exibir o inimigo em um monitor bidimensional. Para isso, o jogo selecionará uma visão estática do mundo chamada portal de visão. Neste caso, a porta de visão está associada ao modelo do player que controlamos. O jogo então usará essa porta de visão para determinar onde um objeto 3D deve ser exibido no mundo.

Uma boa maneira de visualizar sua porta de visão é imaginar uma cena de filme. Ao filmar um filme, o cenário inclui atores, equipamentos de som, equipamentos de iluminação e pessoas e equipamentos responsáveis pelos efeitos reais. No entanto, quando você assiste a um filme, nenhuma dessas informações adicionais será exibida porque a única visão do mundo 3D (cenário) que você pode acessar é a câmera que filmou uma parte específica. Nessa analogia, a câmera serve como uma janela para o mundo 3D de uma cena de filme.

Ao nos mover ao redor do mundo, estamos ajustando a posição da porta de visão. Por exemplo, movendo-se para a direita da posição mostrada na figura, obtemos o seguinte cenário:

Podemos ver que o inimigo não está se movendo, mas seu modelo agora está exibido no lado esquerdo da nossa tela. Isso ocorre porque, quando movemos nossos players, também movemos nossas portas de visão para diferentes locais no mundo.

5.9.5 Mundo para a tela

Assim como fizemos ao desenvolver o aimbot, simplificaremos nosso desenvolvimento ESP isolando primeiro os valores X (ou esquerda e direita). Depois que descobrimos como calcular esse valor, podemos adicionar nosso Y (valores superiores e inferiores). Também desenvolveremos nossos hackers para inimigos individuais e aumentaremos o suporte para vários inimigos.

Neste capítulo, presumimos que você está executando o cubo de ataque na janela 1024x768. Isso significa que nossas janelas têm 1.024 pixels de largura e 768 pixels de altura. Dependendo da localização do seu ponto de vista no mundo, o inimigo aparecerá com um valor de pixel específico ao renderizar a cena. Por exemplo, tomemos como exemplo a cena em que estamos observando o inimigo:

Neste caso, o inimigo está no meio da nossa tela, ou 1024/2. Isso significa que o inimigo está em (aproximadamente) 512º pixel. Quando movermos o jogador para a esquerda, o inimigo agora aparecerá no extremo direito da tela:

É difícil determinar os pixels exatos do inimigo aqui, mas podemos presumir que seja cerca de 1.000. Da mesma forma, se movermos para a direita, o inimigo aparecerá na extrema esquerda da nossa tela:

Os inimigos aqui começam com cerca de 100 pixels.

Podemos usar uma série de equações para representar esses cenários diferentes. Como a parte central da porta de visão da visão padrão está alinhada com a parte central da tela, queremos encontrar um valor S para satisfazer todas as seguintes equações:

```
512=512+S
100=512+segundos
1000=512+segundos
```

Nenhum valor S constante pode fazer com que todas essas equações sejam válidas. Precisamos de uma maneira de tornar S negativo e positivo e representar valores aproximadamente de -400 a 0 a 400. Para conseguir isso, podemos estender S para a multiplicação de dois valores, conforme mostrado abaixo:

```
512=512+(A*F) 100=
512+(A*F) 1000=512+(
A*F)
```

Nessas equações, a estará relacionado à distância do inimigo do centro do nosso campo de visão, e F será um valor de escala estático. Se olharmos diretamente para o inimigo, A será 0, tornando nossa primeira equação válida. Se olharmos para a direita do inimigo, (a*F) produzirá um valor negativo subtraído de 512. Da mesma forma, se olharmos para a esquerda, (a*F) produzirá um valor positivo, adicionado a 512.

5.9.6 valor de zoom

Em seguida, precisamos determinar os valores de A e F. Ao escrever o aimbot, determinamos a curva do jogador atual e a curva necessária para mirar no inimigo. Neste caso, definimos a curva do jogador atual para a última curva. Entretanto, para esse hacker, podemos usar a diferença entre esses valores como o valor acima. Quanto maior a diferença entre esses valores, mais distante o inimigo estará do centro da nossa tela.

Podemos usar nosso código aimbot para determinar como esses valores são no jogo. Em nosso código aimbot, calculamos o giro da seguinte forma:

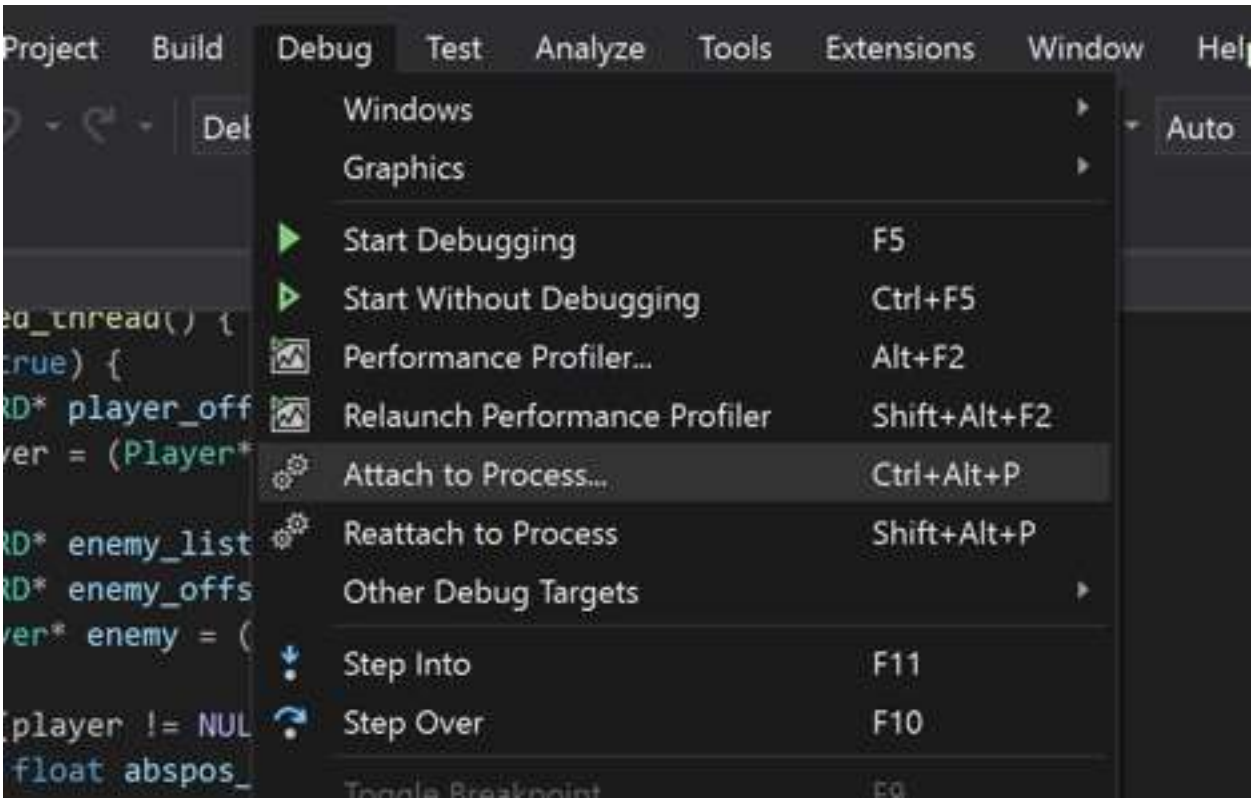
```
float abspos_x=inimigo ->x-player->x;
float abspos_y=inimigo->y-player->y;
```

```
Ángulo de azimute flutuante xy=atan2f (abpos_y,
abpos_x); Cruzamento flutuante = (flutuante) (
ângulo azimute xy*(180,0/M_PI)); Caminho +=90;
```

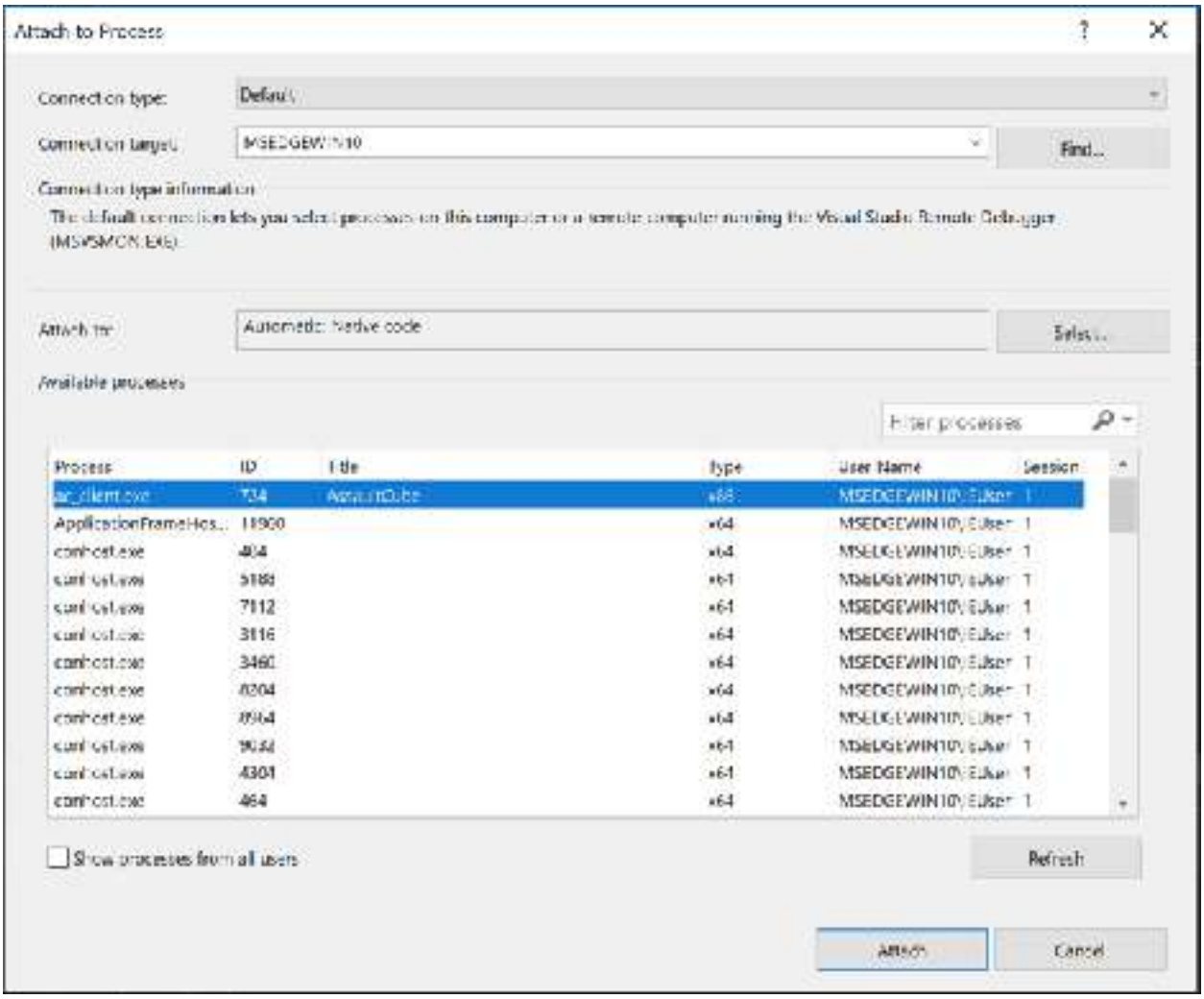
Ao contrário da configuração player ->yaw=yaw no aimbot, calcularemos as diferenças entre essas curvas:

```
float yaw_dif=player->yaw-yaw;
```

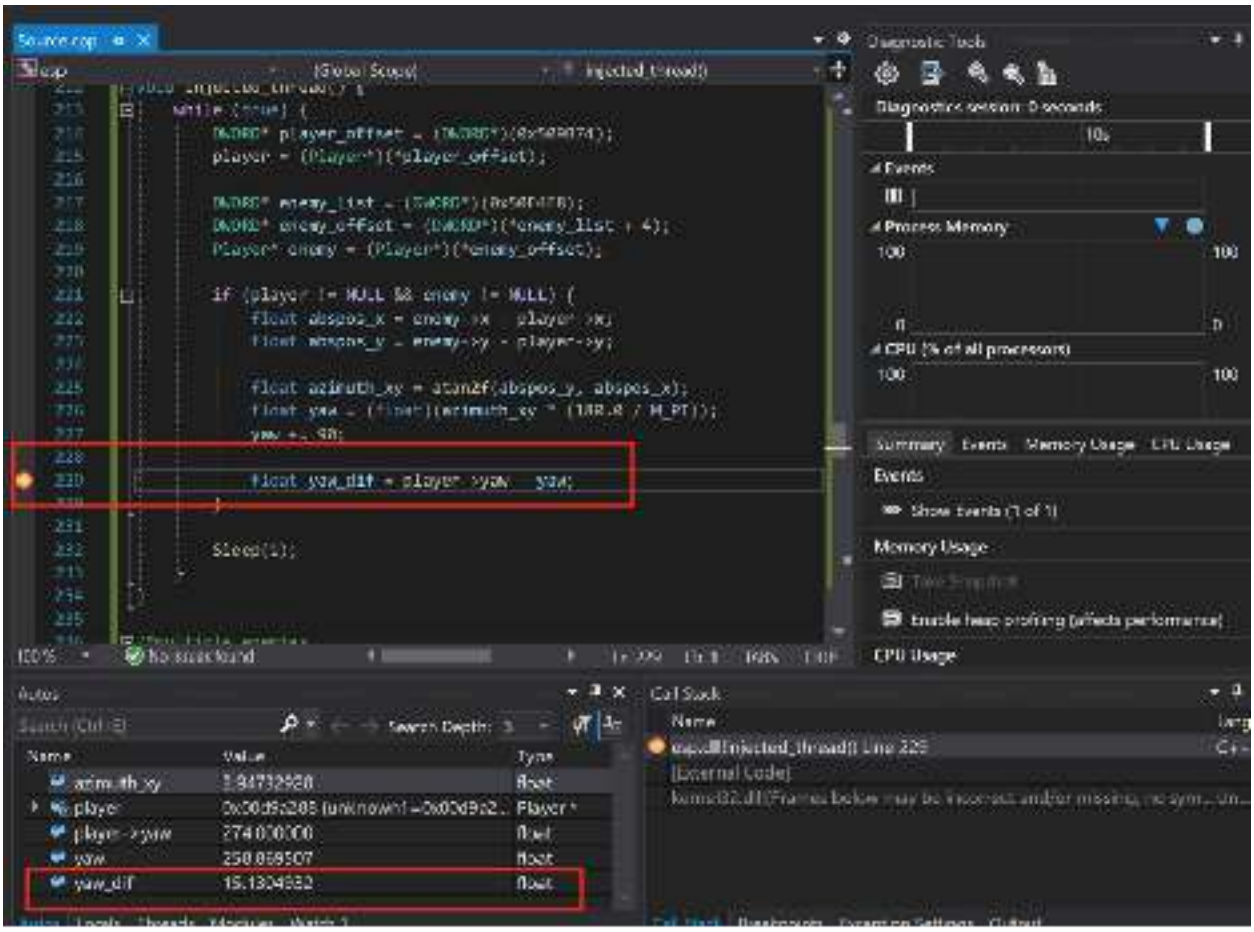
Podemos usar o depurador integrado do Visual Studio para ver qual é esse valor. Primeiro , crie seu DLL como de costume. Em seguida, abra o Cubo de Ataque e crie um jogo para dois jogadores, como fizemos no Capítulo 5.6, injetando DLL com um injetor DLL. Após injetar nosso DLL, entre no Visual Studio e selecione Depurar -> Adicionar ao processo:



Em seguida, selecione o processo de cubo de ataque ac_client:



Quando concluído, você pode definir pontos de interrupção no código DLL, como fizemos anteriormente para código executável regular. Se você colocar o ponto de interrupção na linha especificada yaw_dif, poderá ver seu valor na janela AutoOS na parte inferior do Visual Studio:



Com esta configuração, agora podemos obter o valor yaw_dif. Repita o mesmo cenário que discutimos acima (visualizar, mais à esquerda e mais à direita) e obtenha o valor yaw_dif correspondente a cada cenário:

```
Olhando para o inimigo:
     Caminhada _dif-0.307769775

  Inimigo na extrema esquerda da
       tela: yaw_dif 34.9015427

   Inimigo no extremo direito da
      tela: yaw_dif-39.5185280
```

Seus valores podem variar dependendo de sua localização no mapa. Para fins deste capítulo, usaremos os valores acima como nossas equações. Vamos substituir esses valores na equação como valores de A:

```
Olhando para o inimigo:
              512=512+ (-0.3076975*F)

Inimigo na extrema esquerda da tela:
      <100=512+(34.9015427*F)

Inimigo na extrema direita da tela:
            >950=512+ (-39.5185280*F)
```

Como estamos estimando aproximadamente, arredondamos esses valores para o número inteiro mais próximo:

```
Olhando para o inimigo:
      512=512+(0*F)

Inimigo na extrema esquerda da
      tela: 100=512+(35*F)

Inimigo na extrema direita da
      tela: 1000=512+ (-40*F)
```

Podemos ver que não importa qual valor escolhemos para F, yaw_dif satisfará nossa primeira equação. Usando algumas álgebras básicas, podemos usar as equações mais à esquerda e mais à direita para resolver F:

```
 Inimigo na extrema esquerda da
    tela: F=-11.771428571

Inimigo na extrema direita da
      tela: F=-12.2
```

Como esses são valores aproximados, tomaremos a média solta e selecionaremos -12 como valor de F. A equação inicial para converter a posição do inimigo em coordenadas de tela 2D dimensionais X é:

```
Tela _x=512+ (guia _dif*-12)
```

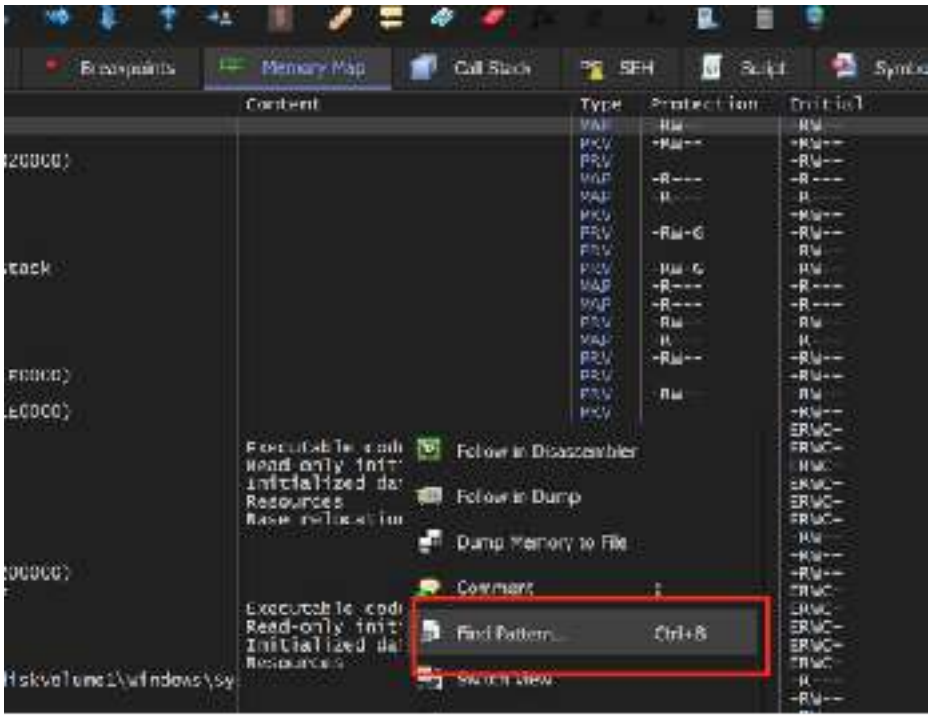
Teremos que ajustar essa equação, mas ela nos dá um bom ponto de partida.

5.9.7 Texto de impressão de localização

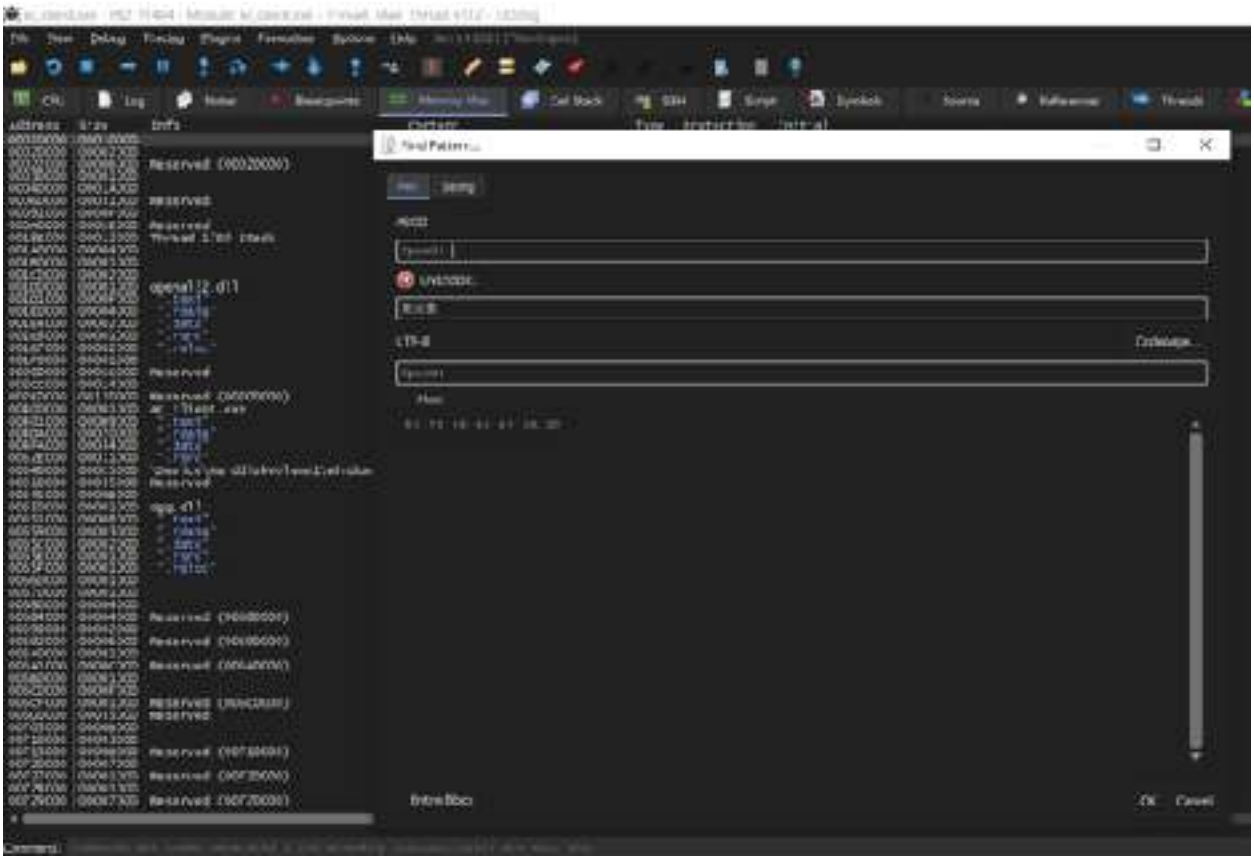
Para continuar testando nossas equações, precisamos encontrar uma maneira de imprimir texto na tela. No Capítulo 3.5, discutimos como encontrar e vincular uma função de impressão de texto. Usaremos uma abordagem semelhante aqui.

Queremos identificar alguns textos que parecem ser facilmente exibidos em qualquer lugar da tela. Depois de estudar alguns documentos, podemos determinar que o texto do comando ShowSpot é um bom texto candidato, pois ele é exibido no meio da tela:

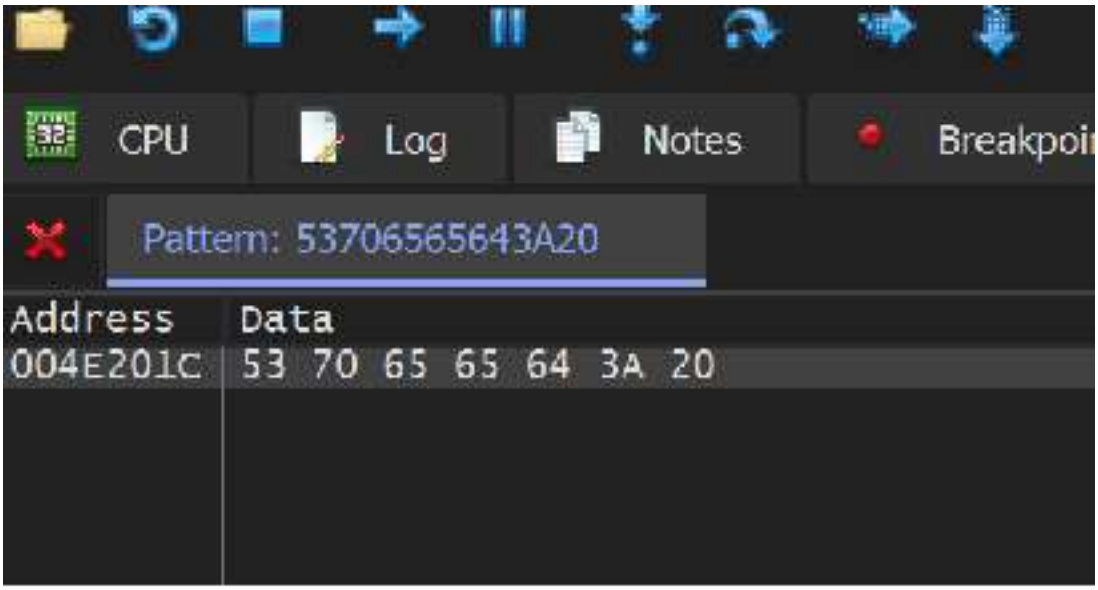
Como esse texto é estático, podemos pesquisar seu padrão em x64dbg e encontrar sua localização na memória. Adicione x64dbg ao assauult Cube, navegue para a aba de mapeamento de memória e clique com o botão direito. Selecione o modo de busca:



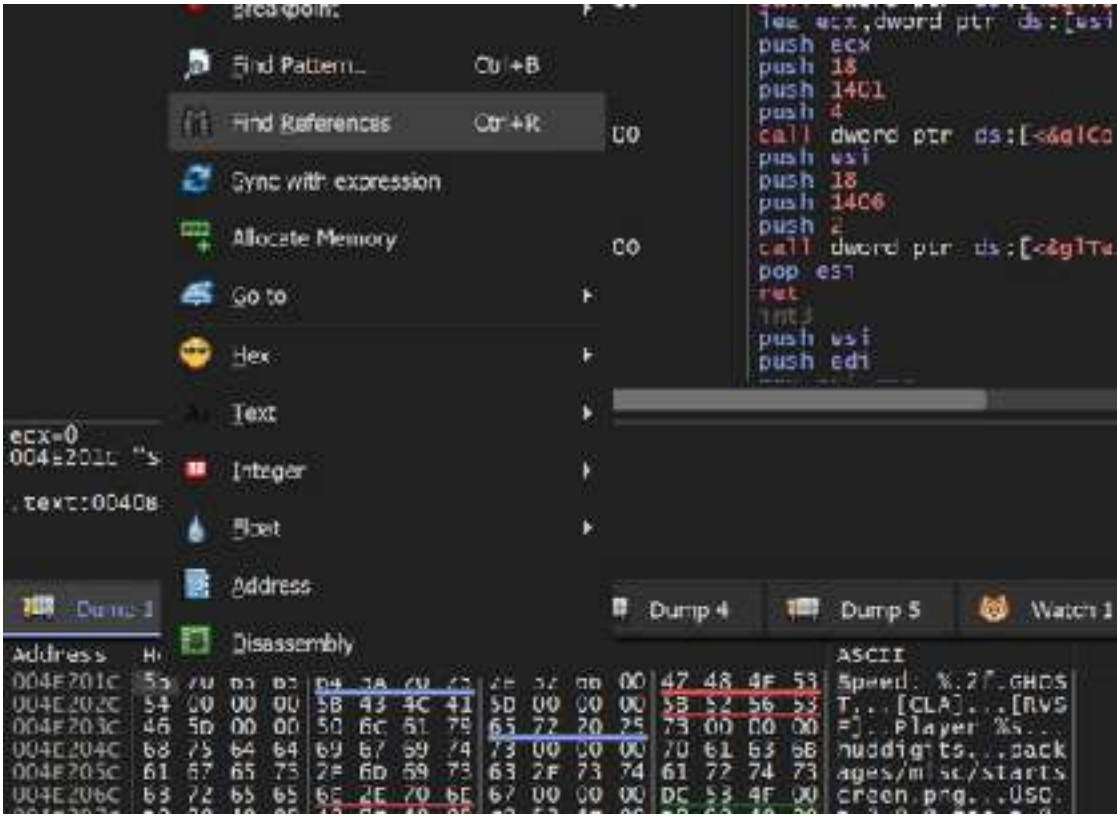
Isso nos permitirá pesquisar por toda a memória ativa do jogo e encontrar qualquer modo de byte que especificamos. Neste caso, pesquisaremos o início deste texto na velocidade:



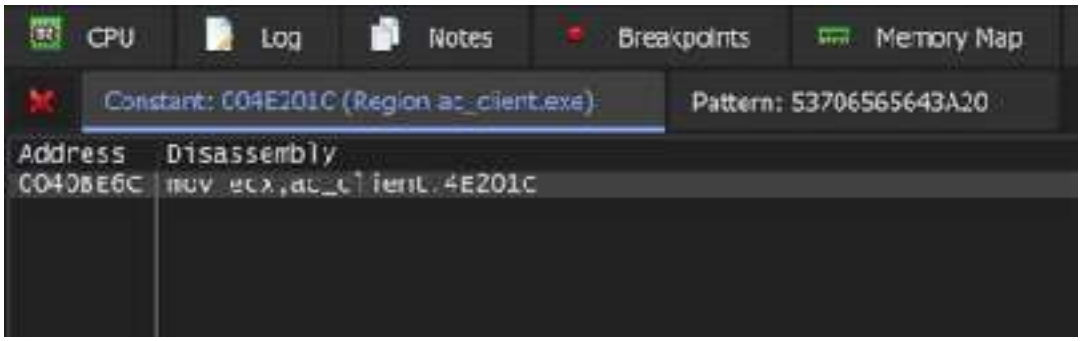
A pesquisa deve encontrar um padrão:



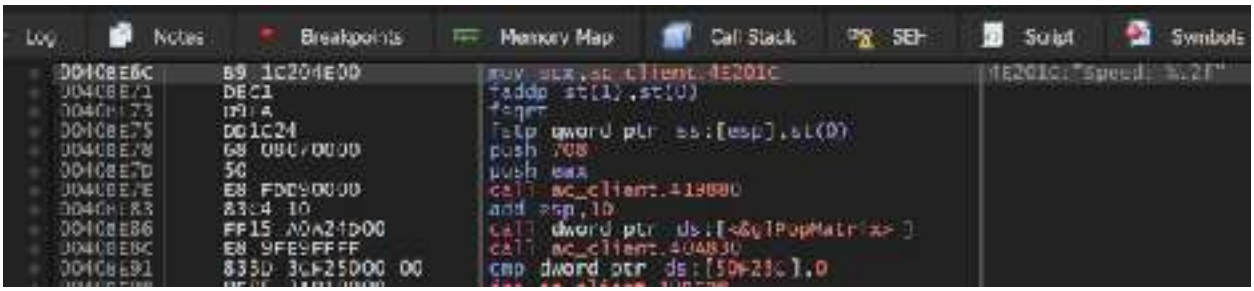
Clique duas vezes nele para exibir o endereço no dump. Em seguida, vamos ver onde esse endereço de memória é referenciado no código. Selecione a primeira letra e clique com o botão direito. Selecione referências de busca:



Isso retorna uma referência, ou o local onde o código do jogo refere este endereço:



Clique duas vezes nesta referência novamente e você será enviado para o código responsável por acessar esta memória:



Analisando rapidamente esse código, vemos que movemos uma string de texto para o ecx (neste caso, nosso espécie: string) e, em seguida, empurramos dois valores para a pilha antes de chamar 0x419880. Podemos ver um valor 0x708, ou 1800 casas decimais . Se definirmos um ponto de interrupção na chamada, podemos ver o valor do eax:

0x4B0, ou 1200 e 1800, parecem ser valores X e Y razoáveis. Combinado com strings de texto em ecx, esta função provavelmente é responsável pela impressão de SPED: text. Podemos verificar esse comportamento alterando push 0x708 para outro valor, como push 0x100:

```
0040BE71    DEC1            faddp st(1),st(0)
0040BE73    D9FA            fsqrt
0040BE75    DD1C24          fstp qword ptr ss:[esp],st(0)
0040BE78    68 00010000     push 100
0040BE7D    50              push eax
0040BE7E    E8 FDD90000     call ac_client.419880
0040BE83    83C4 10         add esp,10
0040BE86    FF15 A0A24D00   call dword ptr ds:[<&glPopMatrix>
```

Após fazer isso, nosso SPED: texto aparecerá perto do topo da tela:

A partir disso, podemos ver que o método responsável pela impressão do texto espera que Y seja empurrado primeiro, seguido por X.

5.9.8 Imprimir código de texto caverna

Para determinar nossa equação, usaremos uma caverna de código para modificar a chamada de texto impresso showspeed para desenhar o texto do nosso inimigo. Vamos enganar o primeiro push para que possamos enviar o valor necessário para a pilha:

```
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

DWORD old_protect;
      unsigned char*hook_localização=(unsigned char*)0x0040BE78;

Se(fdwReason==DL_Process_Attach){
            Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL
, 0, NULL);

            Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
            *Hook_Localização=0xE9;
            *(DWORD*)(hook_localização+1)=(DWORD)&
codecavidade-((DWORD)hook_localização+5);
            *(hook_localização+5)=0x90;
…
```

Nossa caverna de código em si empurra nossos valores X e Y atualmente atribuídos para a pilha e move alguns textos inimigos comuns para o ecx. Após concluir isso, retornaremos à chamada original e pediremos que ela imprima nosso texto:

```
DWORD Ret_Adress=0x0040BE7E; const
char*text= "Espero";

DWORD x=0;
DWORD y=0;

__declspec(naked)void codecave(){_
asm{
mov ecx, push de texto
e
Empurrar x
jmp Ret_ADRESS}

}
```

Depois de calcular yaw_dif, podemos verificar se este código é válido definindo X e Y para 0x100 no thread:

```
x=0x100;
y=0x100;
```

Se entrarmos em um jogo e mostrarmos a velocidade, veremos nosso texto aparecendo no canto superior esquerdo da tela:

Agora, podemos usar isso para determinar nosso ESP. Voltaremos mais tarde para ajustar esse método para que possamos escrever várias strings de texto para vários jogadores.

5.9.9 Equação Refinada

Com a função de texto, agora podemos começar a estudar ESP. No entanto, primeiro precisamos ajustar nossas equações. Quando modelamos a tela pela primeira vez, presumimos que o centro fosse 512. Entretanto, na função de velocidade, vemos que o centro é 0x4B0, ou 1200. Os jogos geralmente usam uma tela "virtual" do mesmo tamanho, independentemente da resolução. Dessa forma, os desenvolvedores só precisam converter a resolução para o tamanho da tela virtual uma vez, mas podem usar coordenadas consistentes para o resto do código.

Nesse caso, a tela virtual do jogo parece ser 2400x1800. Podemos retornar às equações originais e atualizá-las com esses novos números:

```
Olhando para o inimigo:
      1200=1200+(0*F)

Inimigo na extrema esquerda da
      tela: 100=1200+(35*F)

  Inimigo no extremo direito da
      tela: 2400=1200+ (-40*F)
```

Usando esses novos valores para calcular, obteremos valores diferentes para F:

```
Inimigo na extrema esquerda da
tela: F= -31.428571429

Inimigo na extrema direita da
tela: F=-30
```

Como esses também são valores aproximados, selecionaremos -30 como valor de F e , assim, chegaremos à nossa nova equação:

```
Tela _x=1200+ (guia _dif*-30)
```

Podemos implementar essa equação no thread principal, conforme mostrado abaixo:

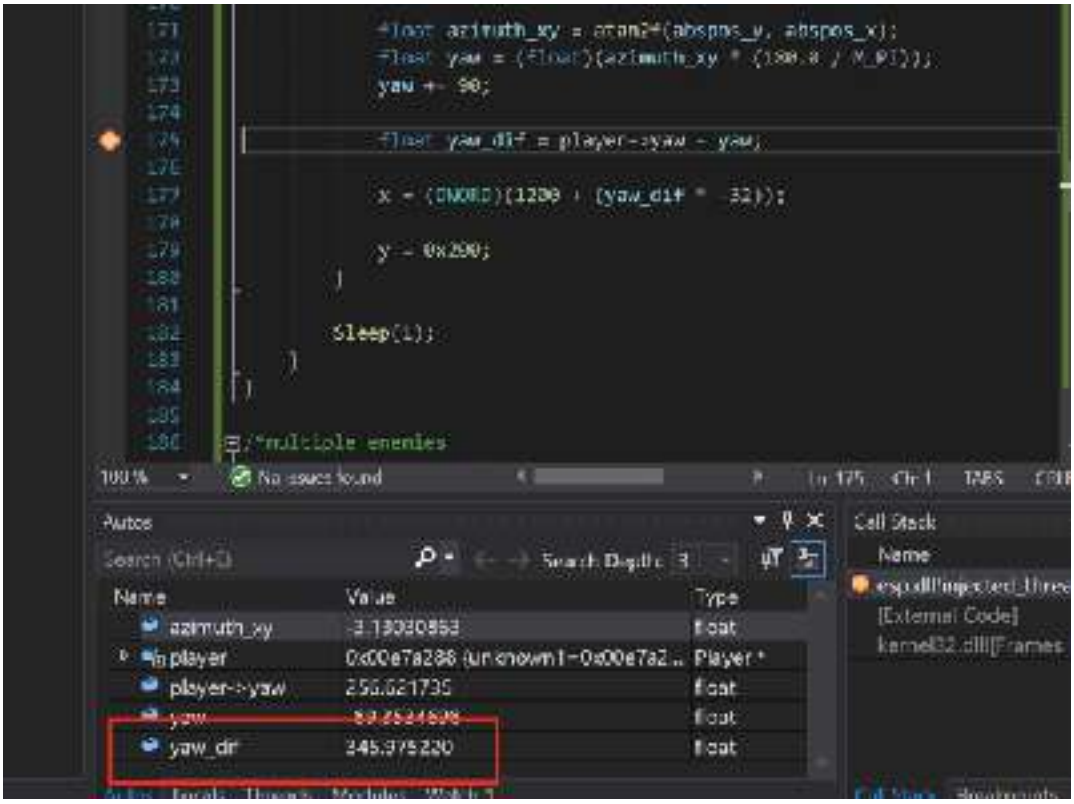
```
float yaw_dif=player->yaw-yaw;
```

```
x=(DWORD)(1200+(yaw_dif*-30)); y=0x
200;
```

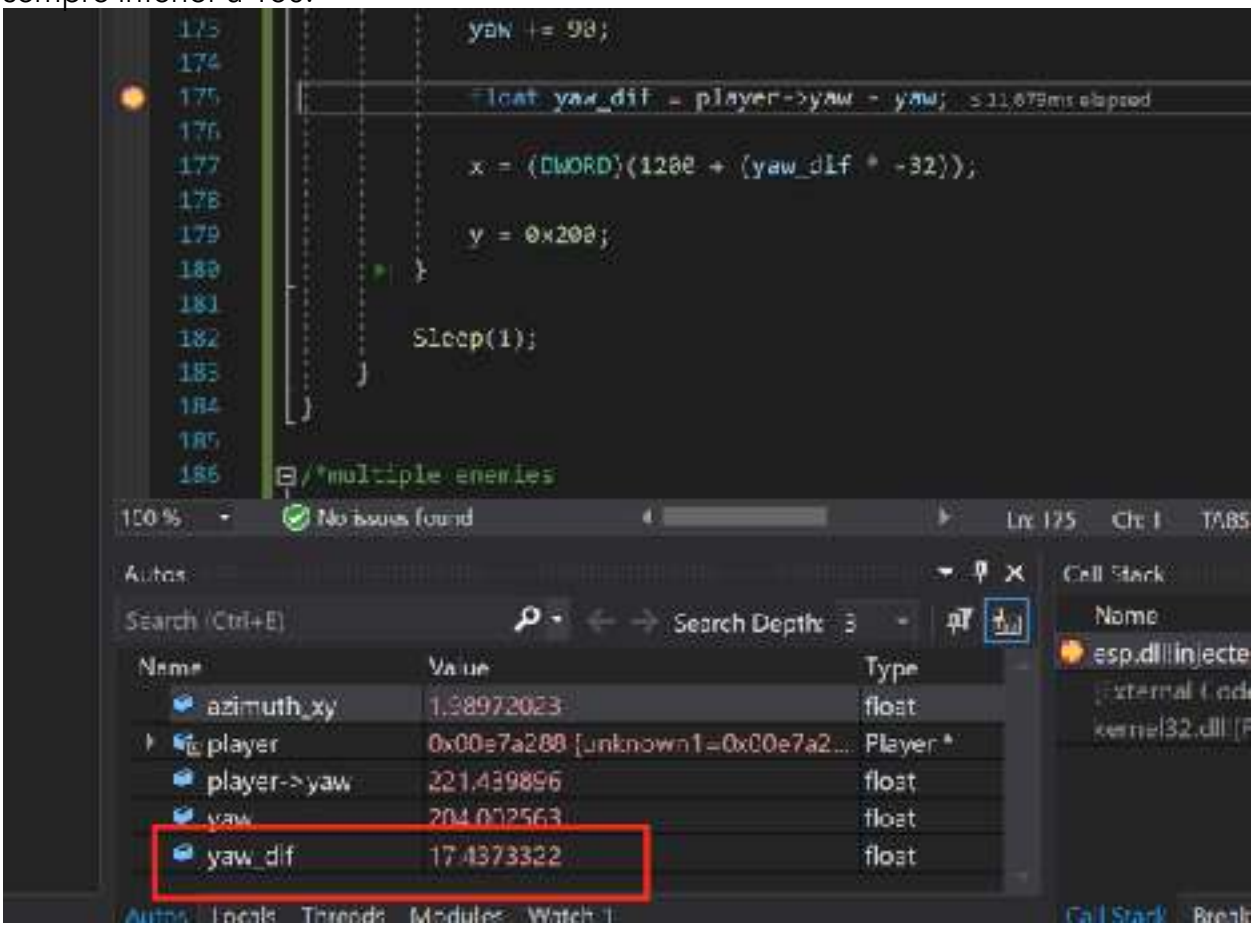
Se você entrar em um jogo, o texto do inimigo aparecerá em certos ângulos no mesmo eixo X do inimigo, como esperávamos:

Entretanto, de acordo com o ângulo em que observamos, o texto aparecerá em um lado distante:

Se adicionarmos um depurador, podemos ver que o valor yaw_dif excede 180 quando o texto está incorreto:

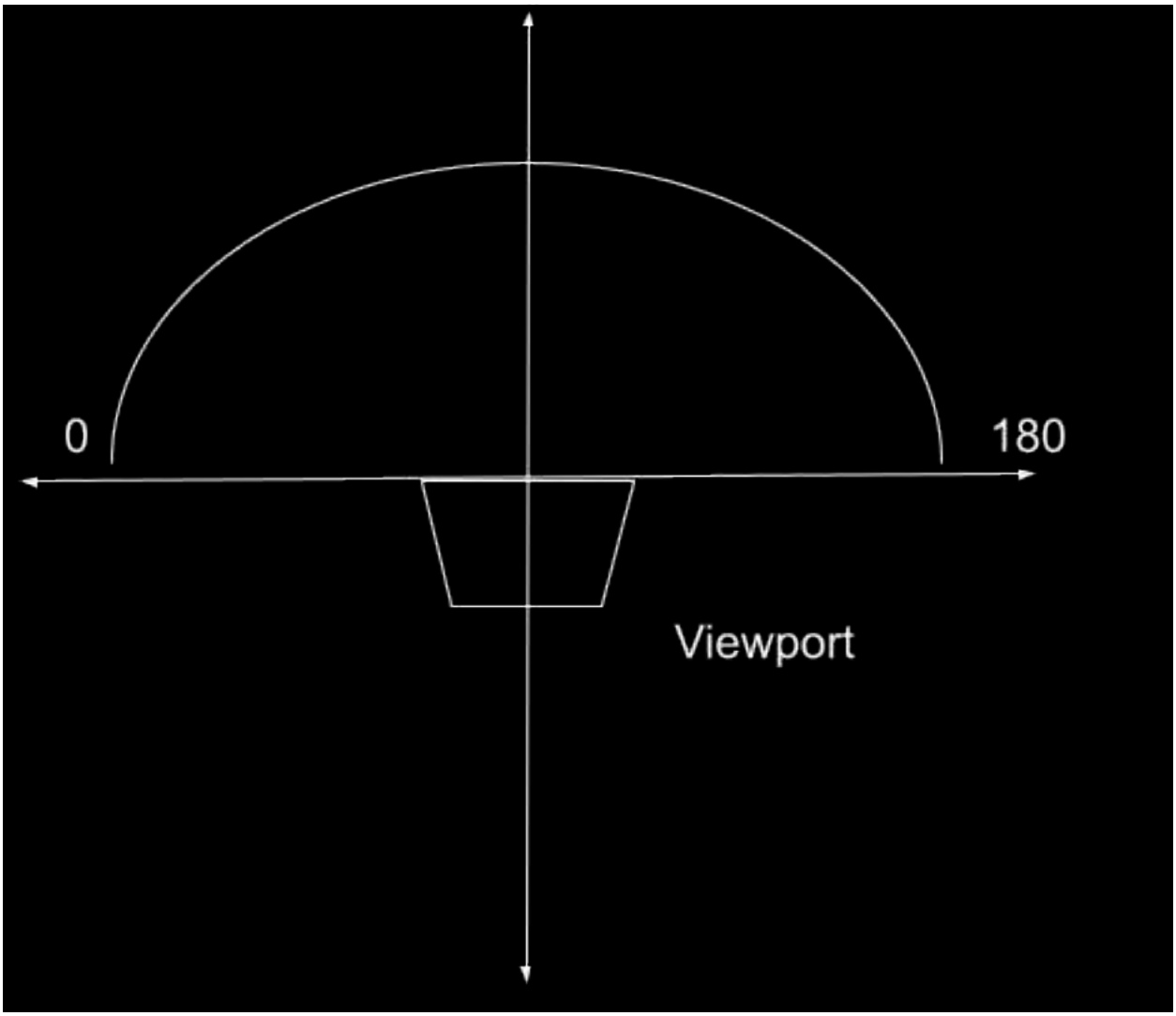


Em comparação, podemos ver que o valor yaw_dif do valor correto do texto é sempre inferior a 180:



Isso ocorre quando o giro do nosso jogador e o giro calculado do inimigo estão entre 275 e 360/0. Quando a subtração obtém nossa diferença, a equação produz valores artificialmente altos que são inconsistentes com nossos fatores de escala. Por exemplo, se nossa margem de guinada for 5, nosso texto será exibido corretamente. Da mesma forma, se nossa diferença for -5, o texto será exibido corretamente na direção oposta. Entretanto, se nossa diferença for -355, o texto será exibido incorretamente porque o resultado da equação agora será 11.850, fazendo com que o texto mude de linha para o outro lado da tela.

Não importa qual porta de visão escolhemos, nossa porta de visão nunca poderá exibir uma tela com mais de 180 graus. Se mais isso nos fará ver atrás dos jogadores.



Para corrigir o caso de -355 que descrevemos acima, podemos subtrair (ou adicionar) 360 no caso de números negativos:

```
Se (guia _dif>180)
      yaw_dif=yaw_dif-360;

Se (guia _dif<-180)
      yaw_dif=yaw_dif+360; x=(DWORD
)(1200+(yaw_dif*-30));
```

Com isso, nosso texto sempre será exibido corretamente, independentemente do ângulo.

Por volta de 5,9.10

Para calcular a dimensão Y do aimbot, temos o seguinte código:

```
float abspos_z=inimigo->Z-jogador->z;
...
Se (abpos_y<0){
abpos_y*=-1;
}
Se (abpos_y<5){
Se (abpos_x<0){
abpos_*=-1;
}
abpos_y=abpos_x;
}
Azimuto flutuante _z=atan2f (abspos_z, abspos_y); Espaço
flutuante = (flutuante) (azimutal_Z*(180,0/M_PI));
```

Podemos usar o mesmo método acima para calcular nossa dimensão Y. Aprendemos pela função de velocidade que 0x708, ou 1800, é a parte inferior da tela virtual. Podemos executar a mesma série de equações acima para obter o fator de escala de Y:

```
  Olhando para o inimigo (inimigo de 1800
       graus) pitch_dif 25.4983654
F=35,4
Olhando para o inimigo, 1800/2
      pitch_dif-4.36527729 f=0

Observe abaixo do inimigo (o inimigo está
      em 100) Pitch_dif-41.125888
F=19,464720195
```

Dentre esses valores, selecionaremos o valor 25 como nosso fator de escalada:

```
float pitch_dif=player->pitch; y=(DWORD)(
900+((pitch_dif)*25));
```

Após concluir isso, o texto agora será exibido corretamente no eixo Y:

5.9.11 Ajuste Final

Agora, nosso texto sempre mostrará, mesmo que o inimigo esteja atrás de nós. Para evitar isso, adicionaremos uma verificação à caverna de código de texto impresso. Nesta verificação, se o inimigo não estiver visível na porta de visão, definimos a string em branco:

```
__declspec(naked)void codecave(){
Se(x>2400||x<0|y<0|y>1800){text="";

}
Caso contrário{
text="inimigo";}
```

Também podemos resolver o problema de que o texto sempre aparece à esquerda do inimigo. Para isso, precisamos apenas adicionar o valor X calculado em 200:

```
Caso contrário{
text="inimigo";}


x+=200;
```

Nosso texto agora é exibido corretamente para um inimigo.

5.9.12 Nome do inimigo

Com nossas coordenadas, podemos exibir os nomes dos inimigos no topo de suas cabeças. Como antes, podemos encontrar o nome do jogador na estrutura do jogador do jogo e adicioná-lo à estrutura do jogador do nosso código. Observando a estrutura do player na memória, vemos que o nome do player está atrás do elemento de guia que identificamos anteriormente:

Se tentarmos mudar o nome do nosso próprio jogador, descobriremos que ele poderia ter até 16 caracteres. Ao subtrair o deslocamento do valor do polegamento, podemos criar um preenchimento com um elemento desconhecido, como fizemos no Capítulo 5.6. A estrutura do nosso player agora parece:

```
Player estrutural{
        Caráter desconhecido 1
        [4]; Ponto flutuante x;
        y flutuante;
        z flutuante;
        Caráter desconhecido 2[
        0x30]; guia flutuante;
        espaçamento flutuante;
        char unknown 3[0x 1dd];
        Nome do caractere[16];
};
```

Agora podemos modificar o código para usar o nome do inimigo em vez do inimigo comum Texto. Primeiro, exclua o código caverna para definir o texto como a condição de "Easy":

```
const char*text="";
...
__declspec(naked)void codecave(){
	Se(x>2400||x<0|y<0|y>1800){text="";

	}

x+=200;
…
```

Em seguida, depois de calcular X e Y, queremos definir o membro do texto como o nome do inimigo atribuindo ponteiros:

```
Texto = inimigo -> nome;
```

Com essa mudança, os nomes dos inimigos agora aparecerão em suas cabeças:

Neste capítulo, exibiremos apenas o nome do inimigo. Métodos semelhantes podem ser usados para exibir armas inimigas, saúde e outras informações.

5.9.13 Vários inimigos

Agora temos um ESP de trabalho para um único inimigo, que podemos expandi-lo para incluir vários inimigos. Assim como fizemos ao criar o aimbot, podemos usar o mesmo código identificado acima e incluí-lo em um loop.

Primeiro, criaremos um array em vez de um valor X, Y e nome. O número máximo de jogadores no jogo Assault Cube é de 32, então usaremos isso como tamanho do nosso array. Para nosso loop, usaremos o número atual de jogadores que determinamos anteriormente, de modo que nenhum elemento adicional do array causará problemas. Como precisamos usar o elemento do player atual no loop de cálculo e no loop de desenho, também criaremos uma variável global para ele:

```
#Definir MAX_PLAYERS 32

DWORD x_values[ MAX _PLAYERS] ={0};
DWORD y_values[MAX_PLAYERS]={0}; char
*NES[ MAX _PLAYERS]  ={ NULL};  int *
currente_Players;
```

Em seguida, modificaremos nossos threads para iterar todos os inimigos da mesma forma que o aimbot. Quando calculamos valores X e Y, armazenamos esses valores em uma matriz, não em um único elemento:

```
Currente_Players=(int*)(0x50F500);

for (int i=1;i <*currente_players;i++){
DWORD*enemy_list=(DWORD*) (0x50F4F8);
DWORD*enemy_offset=(DWORD*) (*enemy_list+(i*4));

...
x_values[i]=(DWORD) (1200+(yaw_dif*-30));
...
y_values[i]=(DWORD)(center_y+((pitch_dif)*25));
...
Nome [i]=Nome do inimigo ->;
```

Por fim, precisamos refazer nossa função de impressão de texto para que possamos imprimir nomes de vários inimigos. Usaremos a mesma localização, mas em vez de substituir os parâmetros push, ganharemos a chamada em si. Em nossa caverna de código, substituiremos as chamadas por texto vazio e criaremos uma função de texto impresso com múltiplas chamadas em loop.

Primeiro, mudaremos a posição do gancho para ganhar a chamada:

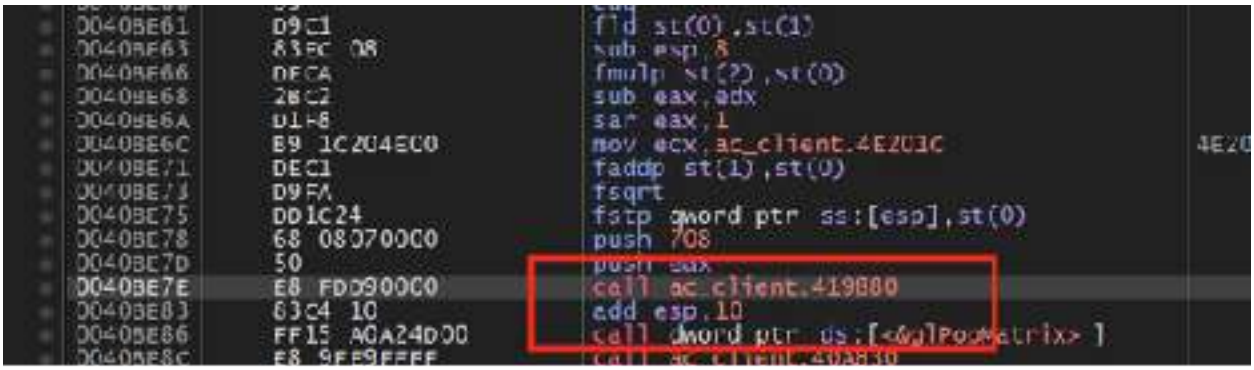
```
unsigned char*hook_localização=(unsigned char*)0x0040BE7E;

Se(fdwReason==DL_Process_Attach){
Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL, 0, NULL);


Projeto Virtuo* (void*)Hook_Locasione, 5. página _Express_Readwrite,
&old_protect);
*Hook_Localização=0xE9;
*(DWORD*)(hook_localização+1)=(DWORD)&codecavidade-((DWORD)hook_localização+5
);
}
```

Podemos então remover a caverna de código anterior e criar uma nova. Nele, primeiro substituiremos o ecx por texto em branco, depois chamaremos a função print text e, em seguida, salvaremos e recuperaremos todo o conteúdo como antes:

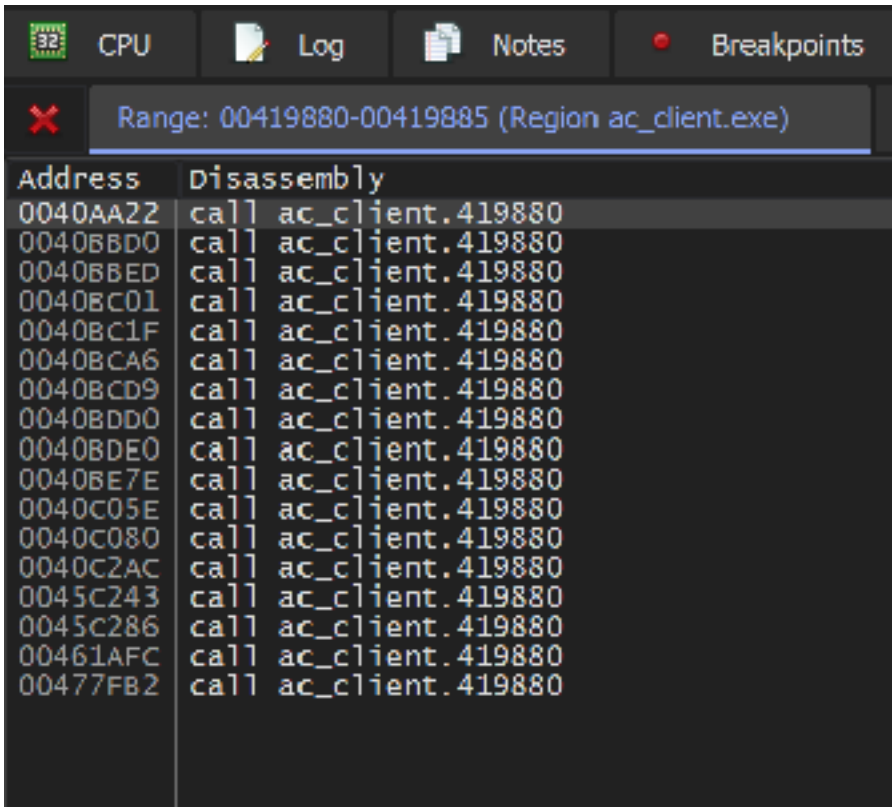
```
DWORD Ret_Adress=0x0040BE83; DWORD
TEXT_ADRESS=0x419880; const char*
empty_text=""; __declspec(naked)void
codecave(){



      _asm{
            mov ecx, empty_text
            chama text_adress push

      }

//Ciclo

_asm{
            Rebote
            jmp Ret_Adress
```

```
        }
}
```

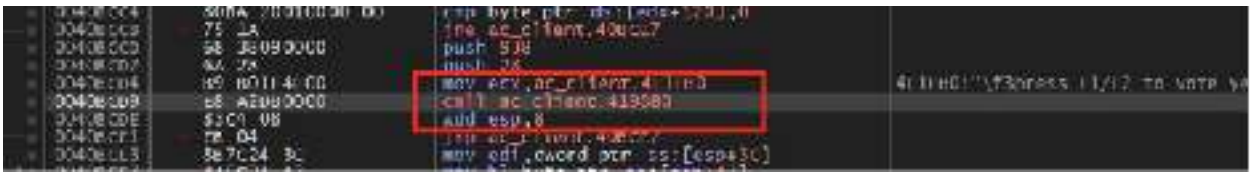
Para chamar a função print text por conta própria, precisamos descobrir como consertar a pilha. Lembre-se do código que encontramos:



Quando você vê um add esp ou sub esp após uma chamada, isso significa que o código chamado quer que você equilibre a pilha. A maneira mais fácil de determinar como equilibrar a pilha é encontrar o valor mínimo do esp que está sendo modificado e usar essa chamada como base. Isso pode ser feito inserindo uma chamada de texto impresso e pesquisando referências:



Depois de navegar por várias dessas referências, você deve encontrar a seguinte referência, que mostra add esp, 8 como o valor mínimo necessário para uma pilha balanceada:



Com essas informações, podemos chamar o texto impresso nós mesmos. Primeiro, criaremos um loop que iterará todos os jogadores atuais do jogo:

```
Currente_Players=(int*)(0x50F500); for (int i
=1;i <*currente_players;i++){
```

Para tornar nossa matriz de valores fácil de usar em blocos asm, copiamos o valor atual para uma variável temporária:

```
const char*text="";

DWORD x=0;
DWORD y=0;
...
Para...
x=x_values[ i]; y=y_
values[i]; text=nomes[
i];
```

Como antes, verificaremos esses valores para garantir que eles possam ser exibidos na tela:

```
Se(x>2400||x<0|y<0|y<0|y>1800){text="";

}
```

Por fim, moveremos o nome do inimigo para o ecx, enviaremos nossos valores X e Y e chamaremos a função de texto impresso. Em seguida, equilibraremos a pilha:

```
_asm{
```

```
mov ecx, push de texto
e
Empurrar x
Ligue para Text_Adress
e adicione esp, 8
}
```

Com isso, agora temos todas as peças necessárias para lidar com vários inimigos. Se você construir um DLL e injetá-lo no ASSULT Cube, verá nomes de vários inimigos aparecerem em suas cabeças:

O código completo está disponível para referência no Apêndice A.

5.10 Múltiplos hackers

5.10.1 Objetivo

Nosso objetivo neste capítulo é atacar o Cubo Mágico 1,2.0,2.

5.10.2 Identificação

Nos capítulos anteriores, criamos vários hackers para o Cubo Mágico, incluindo robôs de gatilho, aimbot e ESP. Neste capítulo, criaremos um multifacetado que combina essas tacas com WallHacks. Depois, adicionaremos um menu interativo que nos permite alternar entre todos os hackers que criamos de uma só vez.

5.10.3 Compreensão

Nos capítulos anteriores, já escrevemos código para Wallhawk, triggerbot, aimbot e ESP. No entanto, se tentarmos juntar todo esse código, adicionar novos recursos logo se tornará imparável. Para criar nosso multihack, usaremos uma técnica de desenvolvimento de software chamada Refactoring.

Ao reconfigurar o código, você usa o código existente e muda sua estrutura sem alterar seu comportamento. Pode haver muitos objetivos na refactorização, mas no nosso exemplo, nosso objetivo é encapsular certas funções em uma classe para que o código possa ser dividido em partes lógicas. Isso limpa o código, tornando-o mais fácil de manter . Quando concluído, construiremos esse código refactorizado para adicionar nosso menu.

Este capítulo cobrirá uma grande quantidade de código. Cada fase separada do código é fornecida no Apêndice A deste capítulo. O código final estará na seção " Completo".

5.10.4 Wall Hacker

No Capítulo 5.3, apresentamos um método para criar um wallhack para jogos usando OpenGL. Podemos usar a mesma técnica no cubo de ataque e fazer algumas pequenas modificações.

No jogo de destino inicial (City Horror), precisamos verificar a contagem e, se a contagem do modelo não for grande o suficiente, precisamos reativar o teste de profundidade. Se não fizermos isso, os testes profundos serão desativados em todos os projetos. Entretanto , na Assault Cube, a lógica de renderização funciona de forma diferente e não é necessária esta verificação. Além disso, o Assault Cube não precisamos nos preocupar em editar aviões. Portanto, nossa caverna de código glDrawElements pode ser simplificada:

```
__declspec(naked)void opengl_codecave(){
	_asm{
		Pushad
	}

	(*glDepthFunc)(0x207);

	//Por fim, restaure o comando original e salte de volta para _
	asm{
		Rebote
		mov esi, dword ptr ds: [esi+0xA18]jmp
		Opengl_Ret_Adress
	}
	}
```

Usando a mesma técnica de gancho descrita no Capítulo 5.3, agora temos um wallhack adequado para cubos de ataque.

5.10.5 Fusão

Os códigos combinados que apresentamos nesta seção podem ser encontrados na seção "Combinações" do Apêndice A.

Nossa primeira tarefa foi mesclar todo o código dos capítulos anteriores em um DLL. Nosso multihack incluirá os seguintes hackers:

- OpenGL Wallhack
- Robô desencadeador
- Abbott
- ESP

Podemos copiar todo esse código em um arquivo principal e alterar qualquer variável ou nome de função em conflito (como em todas as versões do injected _thread). Os resultados são mostrados no Apêndice A.

Ao verificar este código, a primeira coisa que deve aparecer é que ele tem mais de 300 presidentes e 20 variáveis globais. Além disso, podemos ver que temos dois threads criados (um para ganhar o OpenGL e um para o nosso aimbot) e várias cavernas de código.

Usando nosso código combinado, podemos fazer duas pequenas mudanças para aumentar um pouco o tamanho do código. Primeiro, podemos combinar códigos aimbot e ESP porque eles usam a maior parte da mesma lógica. Em segundo lugar, podemos modificar o thread do OpenGL para que ele se desconecte do loop while após conectar o glDrawElements:

```
if(openGLHandle!=NULL&GLDEPTHUNC==NULL){
glDepthFunc=(void(__stdcal*)(sem símbolo)
int)GetProcadodres(openGLHandle,"glDepthFunc");
...
//Como o OpenGL é carregado dinamicamente, precisamos calcular dinamicamente o
endereço de retorno
opengl_ret_address=(DWORD)(OpenGl_Hook_Locação+0x6);}

else{break
;}
```

Depois que ganhamos o OpenGL, isso efetivamente eliminará os threads para evitar que nossos hackers tenham tantos threads abertos.

No formato atual, esse código pode ser criado e usado como múltiplos hackers; No entanto, a manutenção é quase impossível. Se quisermos adicionar um menu e um método para alternar recursos, precisamos verificar cuidadosamente todas as mais de 300 linhas de código e garantir que nossa alternância não introduza nenhum comportamento inesperado em muitos threads. Além disso, atualmente não temos uma boa maneira de imprimir texto fora do ESP.

5.10.6 Primeira Reconstrução

O código-fonte que apresentamos nesta seção pode ser encontrado na seção " Primeira Refactoração" do Apêndice A.

Há muitas maneiras de simplificar nosso código. Para nossos propósitos, encapsularemos as principais funções na classe. Nosso objetivo final é criar classes que possam ser facilmente reutilizadas em outros jogos FPS. Em seguida, chamaremos essas classes do arquivo principal.

Uma classe em C++ geralmente tem dois componentes: o cabeçalho, que descreve o que a classe contém e o que o chamador contém, e o código-fonte, que contém todo o código da classe. Portanto, dividiremos o código do multihack em pastas de cabeça e pastas de origem para todas as seguintes refactoridades.

Trigerbot é um bom ponto de partida. No formato mais básico, sempre que vemos um jogador, nosso Trigerbot envia um evento de mouse para baixo. Para tornar esse código reutilizável, construiremos a classe triggerbot e pediremos ao hacker principal que forneça informações sobre se estamos verificando os jogadores.

Vamos começar com o código Trigerbot atual:

```
__declspec (nua)VoidTriggerbot_Codecave(){__
      asm{
            Chamar Triggerbot_ori_Call_Adress
            push
            mov edi_value, eax}


      Se(edi_valor!=0){
            Entrada. Tipo = mouse de entrada;
            input.mi.dwFlags=MOUSEEVENTF_LEFTDOWN;
            SendInput (1, &input, sizeout (input));
      }
      Caso contrário{

            Entrada. Tipo = mouse de entrada;
            input.mi.dwFlags=mouseeventf_lefutup;
            SendInput (1, &input, sizeof (input));
      }

      _asm{
            Rebote
            jmp trigerbot_ori_jump_adres
      }
}
```

No Assault Cube, o registro edi registra se um jogador está sendo monitorado. No entanto, isso será diferente em outros jogos. Portanto, faz sentido abstrair apenas o código entre blocos _asm. Podemos substituir este código por uma chamada para a classe triggerbot:

```
__declspec (nua)VoidTriggerbot_Codecave(){__
      asm{
```

```
            Chamar Trigerbot_ori_Call_Adress
            push
            mov edi_value, eax
      }

      Trigerbot->Experiência (edi_valor);

      _asm{
            Rebote
            jmp TrigerBot_or_Jump_Adress}

}
```

Agora que sabemos como é o código de chamada, podemos criar classes. Primeiro, podemos criar uma cabeça que contém a definição da classe TrigerBot:

```
#pragma uma vez

#include<Windows.h>

Robô semelhante a gatilho{
Privado:
      Entrada ={0};
Público:
      Trigerbot ();
      void Expectation (int Islooking
Ingenation); };
```

A categoria tem membros privados e públicos. Outros códigos podem acessar membros públicos. Por exemplo, podemos ver que o método execute será chamado diretamente pelo nosso arquivo principal. Entretanto, o arquivo principal não terá acesso às variáveis de entrada.

Para implementar o código da classe triggerbot, criaremos o arquivo-fonte em seguida. Este arquivo conterá o cabeçalho que definimos acima, mas conterá o código real da classe:

```
#include<Windows.h>

#Incluindo "Trigerbot.H"

 Trigerbot:: Trigerbot(){
        input={0};
```

```
}

void TrigerBot::: Expressure (int Islookingatemy)
      {if(int Islookingatemy! =0){
            Entrada. Tipo = mouse de entrada;
            input.mi.dwFlags=MOUSEEVENTF_LEFTDOWN;
            SendInput (1, &input, sizeout (input));
      }
      Caso contrário{
            Entrada. Tipo = mouse de entrada;
            input.mi.dwFlags=mouseeventf_lefutup;
            SendInput (1, &input, sizeout (input));
      }
      }
```

Este é o mesmo que o código original do triggerbot, exceto que agora está encapsulado nesta classe. Podemos adicionar essa classe ao hack de outro jogo. Supondo que o arquivo de fonte primária naquele hack forneça o valor correto para o Islookingatemy, ele funcionará.

Para usar essa classe no código principal, precisamos incluir o cabeçalho e criar uma instância dele:

```
#Incluindo "Trigerbot.H"
...
Robótico* Robótico* Robótica acionada;
...
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){


      if(fdwReason==DL_Process_Attach){
            Trigerbot=Novo Trigerbot();
      }
      else if (fdwReason==DL_Process_Detach){
            delete Trigerbot;;;
      }
```

Ao construir o código dessa maneira, nosso código TrigerBot agora é simplificado e facilmente modificado. Por exemplo, para ligar e desligar o Trigerbot, podemos simplesmente adicionar uma condição, como:

```
Se (trigerbot_enabled){
Trigerbot->Experiência (edi_valor);
```

```
}
```

Também podemos aproveitar esta oportunidade para remover a entrada de variáveis globais do arquivo principal.

5.10.7 Reconstrução concluída

O código desta seção está na seção "Refactoring Completo" do Apêndice A. Outro componente importante que queremos separar é o código responsável pelo aimbot e ESP. Visando o código, vemos que ele é responsável por definir os seguintes valores:

```
x_values[i]=(DWORD)(1200+(yaw_dif*-30)); y_
values[i]=(DWORD)(900+((pitch_dif)*25); Nome [i]
=Nome do inimigo ->;

Player -> Jame = Jame mais próximo
; player->pitch=closest_pitch;
```

Os valores X, Y e Nome são usados para ESP, enquanto os giros e inclinações do jogador são usados para aimbot. Para calcular esses valores, nosso aimbot e ESP exigem a base do jogador, a base da lista de inimigos e o número atual de jogadores no jogo.

Para encapsular esse comportamento em uma classe, dividiremos a função em duas funções. A primeira função será responsável por calcular todos os valores X, Y e nomes do ESP, bem como os curvas e pitch mais recentes. A segunda função será responsável por definir a visão do jogador para a posição calculada. Separar esses recursos permitirá alternar facilmente entre ESP e aimbot.

Como essa classe é responsável pela geometria do jogador e pela relação entre o jogador e o mundo, nós a chamamos de players. Assim como usamos o Trigerbot, podemos alterar o thread do aimbot para:

```
void AIMBOT_THEAD(){
      while(true){
Player Geometry->update(); player Geometry-
>set_player_view();}
```

```
}
```

Como esse código agora é fácil de manter, podemos combinar dois threads no arquivo principal:

```
void injectid_THREAD(){

      while(true){
...
                        if(openGLHandle!=NULL&GLDEPTHUNC==NULL){
...
            }

            Player Geometry->update(); player
            Geometry->set_player_view(); Sono (
            1);

      }
      }
```

Nossa classe Player Geometry incluirá todas as funções relacionadas ao jogador. Para processar a impressão de ESP no arquivo principal, esta classe divulgará matrizes de valores X, Y e nomes:

```
Qualquer player Geometry{
Privado:
      DWORD Play_Offset_Adress;
      DWORD Enemy_list_Adress;
      DWORD Currente_Players_Adress;

      closest_yaw flutuante;
      closest_pitch flutuante;

      Jogadores * Jogadores;

      float Euclidea_Dostance (float, float);
Público:
      DWORD x_values[MAX_PLAYERS]={0};
      DWORD y_values[MAX_PLAYERS]={0}; char
      *NES[MAX_PLAYERS]={NULL}; int*
      currente_Players; Geometria do
      jogador (DWORD, DWORD, DWORD);
```

```
      void update();
      void se_Player_VIEW();
};
```

Ao contrário da classe Trigerbot, que requer apenas um parâmetro, esta classe requer o endereço do jogador, o endereço da lista de inimigos e o número atual de jogadores no jogo. Passaremos isso no construtor da classe, uma função especial que é executada quando a classe é criada:

```
Player Geometry:: Player Geometry (DWORD P_Adress, DWORD E_Adress, DWORD Cp
_Adress){
      Player_Offset_Adress=P_Adress; enemy_
      list_address=e_address; Currente_
      Players_Adress=cp_Adress;
}
```

Podemos então usar esses valores no código da classe:

```
void Player Geometry:: update(){
      DWORD*play_Offset=(DWORD*) (play_offeset_adress); player
      * (player*)(*player_offset);
... O resto do código aimbot e ESP...}


void player Geometry::: set_player_view
      (){player->yaw=closest_yaw; player
      ->player=Closest_Player;
}
```

Quando criamos esta classe no arquivo principal, passamos esses valores. Isso nos permite reutilizar o código aimbot em qualquer jogo com um layout de memória semelhante:

```
Player Geometry = Novo Player Geometry (0x 509b 74, 0x 50f 4f 8, 0x 50f 500);
```

Também precisamos ajustar o código ESP para usar valores nesta categoria:

```
for (int i=1; i<*playerGeometry->current_players;i++) {x=
playerGeometry->x_values[i];
y=playerGeometry->y_values[i]; text=player geometry->Nomes[i
];
```

Por fim, podemos mover algumas variáveis que nunca mudam para a cabeça constante, simplesmente para separá-las do arquivo principal.

5.10.8 Adicionar menu

O código para o restante deste capítulo está na seção "Completo" do Apêndice A. À medida que o código é refactorizado, podemos adicionar um menu. Primeiro, extraímos a função de impressão de texto em suas próprias funções:

```
void Priment_Text(DWORD x, DWORD y, const char*text){if(
      x>2400||x<0|y<0|y<0|y>1800){
            text=";
      }

      x+=200;

      _asm{
            mov ecx, push
            de texto e
            Empurrar x
            Ligue para Text_Adress
            e adicione esp, 8
      }
      }
```

Assim como fizemos na reconstrução, colocaremos a função do menu em sua própria classe. Nosso menu precisa lidar com duas coisas:

- Abra e feche projetos

Exibir o cursor e um conjunto de itens de menu

Vamos primeiro nos concentrar no menu de exibição. Para facilitar a exibição do menu, criaremos dois array na definição da classe do menu: um array contém o texto de exibição do item e o outro contém o status do item:

```
#Definir MAX_ITEMS 4

Público:
const char*items[MAX_ITEMS]={"Wallhack", "ESP", "Aimbot", "Trigerberbot"};
```

```
bool ITEM_EBLEDs[MAX_ITEMS]={false};
```

Também precisamos de uma maneira de retornar uma sequência de On ou OF com base no status do projeto:

```
const car*Menu::: get_estante(intet){return itet
      _enabled[item]? "On": "Off";
}
```

Com essas seções, agora podemos adicionar um loop à caverna de código de texto para exibir todos os itens de menu:

```
for(int i=0;i <MAX_ITEMS;i++){
Priint_Text (50.250+(100*i), menu ->Item [i]); Priint_Text (
500,250+(100*i), menu ->get_state(i));}
```

Depois de imprimir o projeto, podemos continuar adicionando o cursor. Nosso cursor precisa de um caractere e uma posição, então adicionaremos isso à definição de classe. Também precisamos criar uma função externa para processar todas as entradas no menu:

```
Público:
Posição do cursor int;
const char*cursor=">"; const
char*get_Stado (int);
```

Para processar nossas entradas, usaremos o GetAsyncKeyState, semelhante ao que fizemos nos capítulos anteriores. Primeiro, lidaremos com o seguinte:

```
void Menu:: handle_input(){

      if(GetaseNCK_Down)&1){Cursor_Posição++;

      }
      else if(GetaseNCK_UP)&1){Cursor_Posição
            --;
      }
```

O efeito da &1 é registrar um botão apenas uma vez em um curto período de tempo, em vez de enviar spam. A documentação da API discute esse comportamento.

Se pressionarmos para a esquerda e para a direita, queremos ativar ou desativar um projeto. Como todos os estados do projeto são verdadeiros ou falsos, podemos simplesmente alternar seus valores atuais usando o operador not (!):

```
else if((VK_LEFT)&1)||(VK_LEFT)(VK_LEFT)&1)){

item_enabled[Cursor_Posição]=! item_enabled[Cursor_Posição];}
```

Se navegarmos pelo limite do menu, queremos que o cursor apareça na outra extremidade. Podemos fazer isso adicionando algumas verificações:

```
Se (Cursor_Posição<0){
Cursor_Posição=3;
}
Caso contrário, se (Cursor_
Posição>3){Cursor_Posição=0;
}
```

Agora podemos adicionar o menu ao arquivo principal. Primeiro, precisamos do código de texto cave para imprimir o cursor. Deslocaremos a posição atual em múltiplos de 100 para fazer com que ela pareça estar se movendo:

```
Priente_Text (10,250+ (100*menu->Cursor_Posição), menu->cursor);
```

Em nossos threads, também precisamos passar a entrada para o menu para verificar as teclas:

```
Menu->handle_input();
Player Geometry ->update ();
```

Função de comutação 5.10.9

Por fim, precisamos alternar a função de acordo com o status do menu da função. Criamos uma matriz de todos os estados de itens. Para determinar o valor atual de um dos recursos, podemos consultar este array da seguinte forma:

```
if(menu ->ittem_enabled[0])
```

Para tornar essas entradas mais legíveis, podemos criar uma constante no título do menu para referir à localização de cada item:

```
#Definir WALLHACK 0
#Definir ESP 1
#Definir AIMBOT 2
#Definir Trigerbot 3
```

Podemos verificar em nosso código usando esses valores. Por exemplo, para alternar para wallhack, podemos alterar o código para:

```
if(menu->ittem_enabled[wallhatk]){(
*glDepthFunc)(0x207);
}
```

Isso se baseia em nosso trabalho anterior de reconstrução. Para alternar nosso aimbot, podemos fazer facilmente as seguintes operações em um thread:

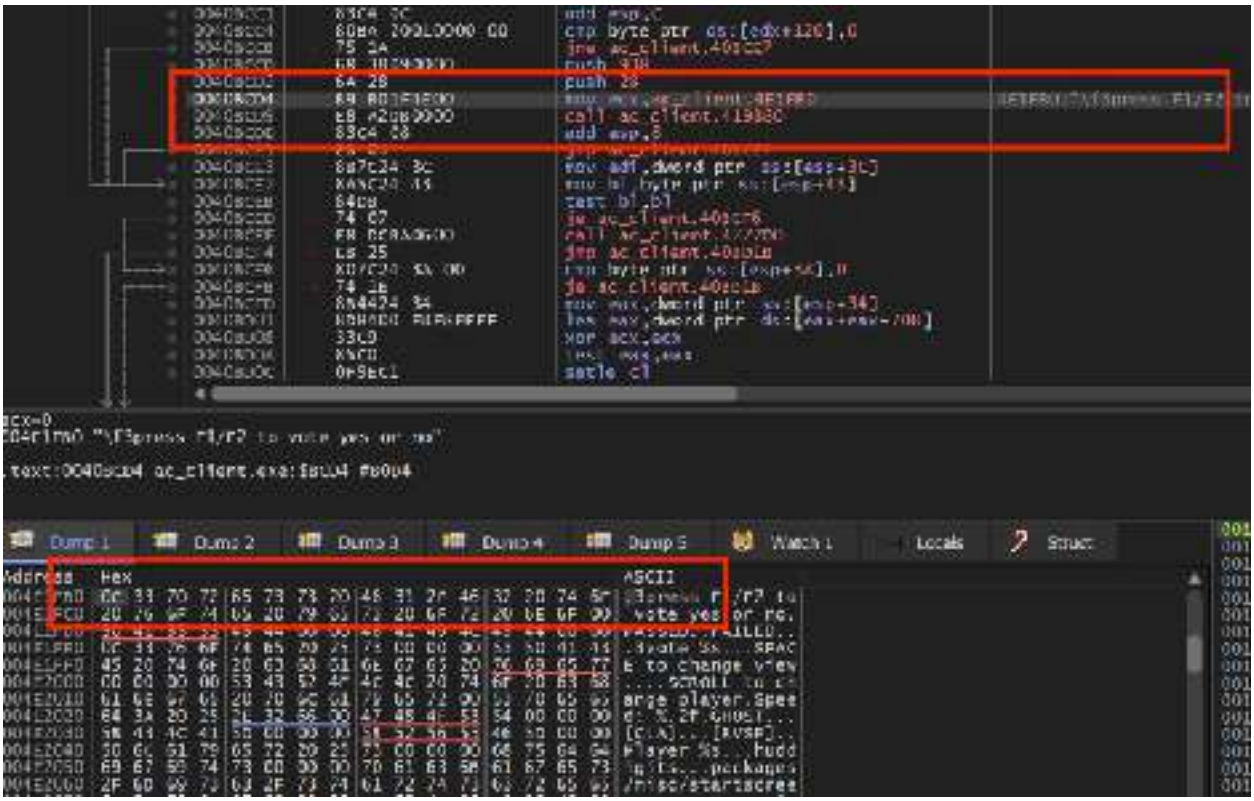
```
if(menu->item_enabled[AIMBOT]){
Playgeometry->se_play_VIEW();}
```

Verificações semelhantes podem ser realizadas no triggerbot e no ESP.

5.10.10 Adicionar cor

Só por algum talento visual, podemos adicionar cor aos itens de menu para torná-los mais fáceis de ler. Ao enviar votações no jogo, podemos ver que alguns textos no jogo já foram

Cor, como texto de votação F1. Se verificarmos esta string em x64dbg, veremos os seguintes dados:



Parece que a string com prefixo 0x0C 33 recebeu uma cor. Se procurarmos por outras strings, veremos que 0x0C está sempre lá, mas ocasionalmente o valor de 0x33 pode ser diferente. Vamos mesclar esses bytes em uma string ligada/desligada:

```
Menu de classe{
Privado:
      const char on_text[5]={0xc, 0x33, 'O', 'N', 0}; const char off
      _text[6]={0xc, 0x33, 'O', 'F', 'F', 'F', '0};
```

Se você entrar no jogo, verá que ambas as nossas cordas agora estão vermelhas. Se você tentar 0x33 e experimentar valores diferentes (0x31, 0x38, etc.), você verá que 0x30 é verde. Agora podemos modificar o código para alterar o texto On para verde:

```
const char on_text[5]={0xc, 0x30, 'O', 'N', 0};
```

Com isso, nossos múltiplos hackers estão concluídos.

Parte 6

Jogos multijogador

6.1 Jogos Multijogador Fundamentos

6.1.1 Equivalência - 2 Equivalência

Imagine dois vizinhos querendo jogar um jogo de xadrez. Uma maneira pode ser pedir ao vizinho A que monte um tabuleiro de xadrez em sua casa e dê ao vizinho B a chave da casa. O vizinho A pode agir em qualquer momento do dia. Entretanto, se o vizinho B quiser agir, ele tem que ir até a casa do vizinho A. Além disso, se o vizinho B quiser pensar enquanto se move, ele precisará tirar uma foto ou registrar uma cópia do tabuleiro de xadrez de alguma forma antes de retornar à sua casa.

Este é um exemplo de um modelo de pares para pares (P2P). No modelo P2P, um jogador atua como console e todos os outros jogadores atuam como hóspedes. Este é um modelo usado por muitos jogos de console para lidar com funcionalidades de jogos multijogador. Sua principal desvantagem é que o host tem uma vantagem em termos de tempo de resposta. Isso ocorre porque todos os outros jogadores devem se conectar ao console para recuperar e enviar atualizações, que podem atualizar suas cópias locais.

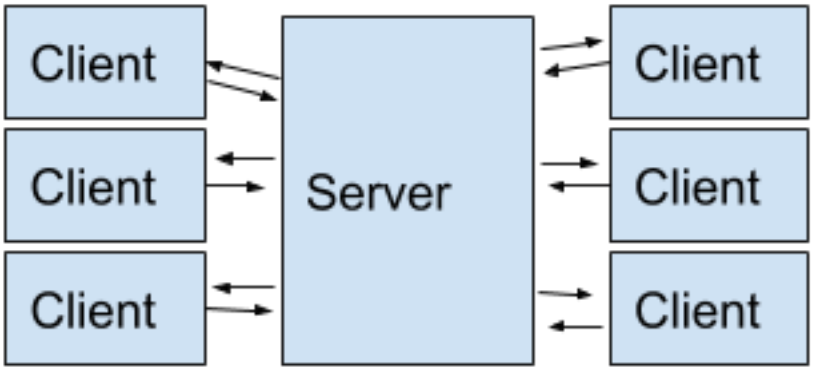
6.1.2 Cliente-Servidor

Agora imagine que os vizinhos A e B querem jogar xadrez, enquanto o outro vizinho (C) quer observar o jogo. Desta vez, cada vizinho tem seu próprio tabuleiro de xadrez, e todos os jogadores concordam que um vizinho extra (D) será o juiz. O juiz é o jogador mais confiável de todos os participantes. Seu papel é garantir que nenhum movimento ilegal ocorra e manter a versão "correta" do tabuleiro.

Para se mudar, o vizinho A escrevia seu histórico de mudança em um envelope e depois o colocava na caixa de correio do vizinho D. O vizinho D então garantirá que o celular seja legal, atualizará seu quadro de avisos e colocará as cartas contendo celular nas caixas de correio dos vizinhos B e C. Os vizinhos A, B e C são responsáveis por atualizar seu tabuleiro de xadrez para corresponder ao tabuleiro de xadrez do vizinho D. Se o movimento enviado pelo vizinho A for

É impossível, o vizinho D o avisaria de que seu conselho de administração não parecia estar atualizado e que ele não poderia agir.

Este é um exemplo de um modelo cliente-servidor, que discutimos brevemente no Capítulo 1.2.



No modelo cliente-servidor, o servidor é uma entidade confiável à qual todos os clientes estão conectados. Ao jogar jogos multijogador, o servidor não participa diretamente do jogo, mas é responsável por salvar cópias confiáveis do status do jogo . Cada cliente envia atualizações ao servidor, que as distribui para outros clientes. Se um jogador enviar muitas atualizações ilegais, o servidor avisará o cliente de que ele não está sincronizado antes de iniciá-lo.

6.1.3 Pacotes de dados

No exemplo do xadrez cliente-servidor, cada vizinho coloca um envelope na caixa de correio do vizinho D contendo seu celular. Na rede, esses pacotes são chamados de pacotes de dados. Assim como um envelope, um pacote de dados contém de quem vem o pacote de dados, para quem o pacote de dados vai e os próprios dados. Por exemplo, se um jogador enviar uma mensagem de bate-papo de olá em um jogo multijogador como o Wesnoth, o pacote de dados pode conter as seguintes informações:

```
Fonte: Destino do
jogador: Dados do
servidor: Olá
```

Quanto maior o pacote de dados, mais tempo leva para ser transferido do cliente para o servidor. Quanto mais tempo você gasta, maior será o atraso na comunicação entre cliente e servidor. Para garantir latência mínima, o pacote de dados contém o mínimo de informações possível. Por exemplo, se um cliente quiser dizer que disparou uma arma, o pacote de dados pode parecer:

```
Fonte: Destino do
jogador: Dados do
servidor:
```

Neste exemplo, tanto o cliente quanto o servidor concordam que f representa um tiro e 1 representa uma rodada.

6.1.4 Protocolo de Rede

Se você quiser dizer a alguém: "Eu gosto deste restaurante", você precisa ter certeza de que está falando a mesma língua. Dependendo da língua, a frase pode ter sintaxe ou estrutura diferente, ou certas palavras podem ser diferentes. A mesma lógica se aplica quando pacotes são enviados pela rede. Essas regras de comunicação são chamadas de protocolos. Esses protocolos determinam como a fonte e o destino se comunicam e como um único pacote de dados aparece. Os dois protocolos principais que você encontrará ao verificar o tráfego da rede do jogo são UDP e TCP.

Imagine que você está tentando enviar uma carta para um vizinho do outro lado da rua lembrando-o de regar suas plantas. Você não esperava uma resposta para esta carta, então a entregou ao seu cachorro. Você espera que esta carta possa ser entregue a ele, mas se não, você não ficará particularmente triste. Este é um exemplo de um UDP, onde não há maneira de confirmar que eles chegaram ao enviar pacotes de dados.

Agora imagine que você quer trocar várias cartas com seu vizinho. Como você responderá diretamente ao que seu vizinho diz, você terá que garantir que todas as cartas sejam entregues. Quando você recebe uma carta, você e seus vizinhos concordam em acender suas respectivas luzes de varanda. Este é um exemplo de um TCP, onde uma conexão frontal é estabelecida e o pacote de dados é confirmado como tendo sido passado.

Os dados contidos em pacotes TCP e UDP podem ser os mesmos, mas os pacotes serão diferentes. Isso ocorre porque cada protocolo tem um cabeçalho diferente, e tanto a fonte quanto o destino usam esse cabeçalho para entender os dados no pacote.

6.1.5 Tomada

Tanto os pacotes de dados TCP quanto UDP usam o Protocolo de Internet (IP) para lidar com o processo de roteamento dos pacotes de dados da fonte para o destino. Cada dispositivo de rede tem um endereço IP que indica a "localização" do dispositivo.

Tráfego (como navegação na web, e-mail ou bate-papo por vídeo), pacotes de dados também têm números de porta. Por exemplo, para navegar pelo site via HTTP, você pode acessar 123.45.67.89 na porta 80. Ao navegar, você também pode se conectar à máquina usando SSH (outro serviço) na Porta 22. Ambas as solicitações podem ser processadas simultaneamente porque são processadas por diferentes programas que monitoram portas diferentes.

IP: Os pares de portas às vezes são chamados de socket. O socket representa o ponto final com o qual ele pode se comunicar. O Windows tem uma API chamada WinSock, que permite aos programadores escrever rapidamente programas que se comunicam com diferentes destinos por meio de TCP ou UDP.

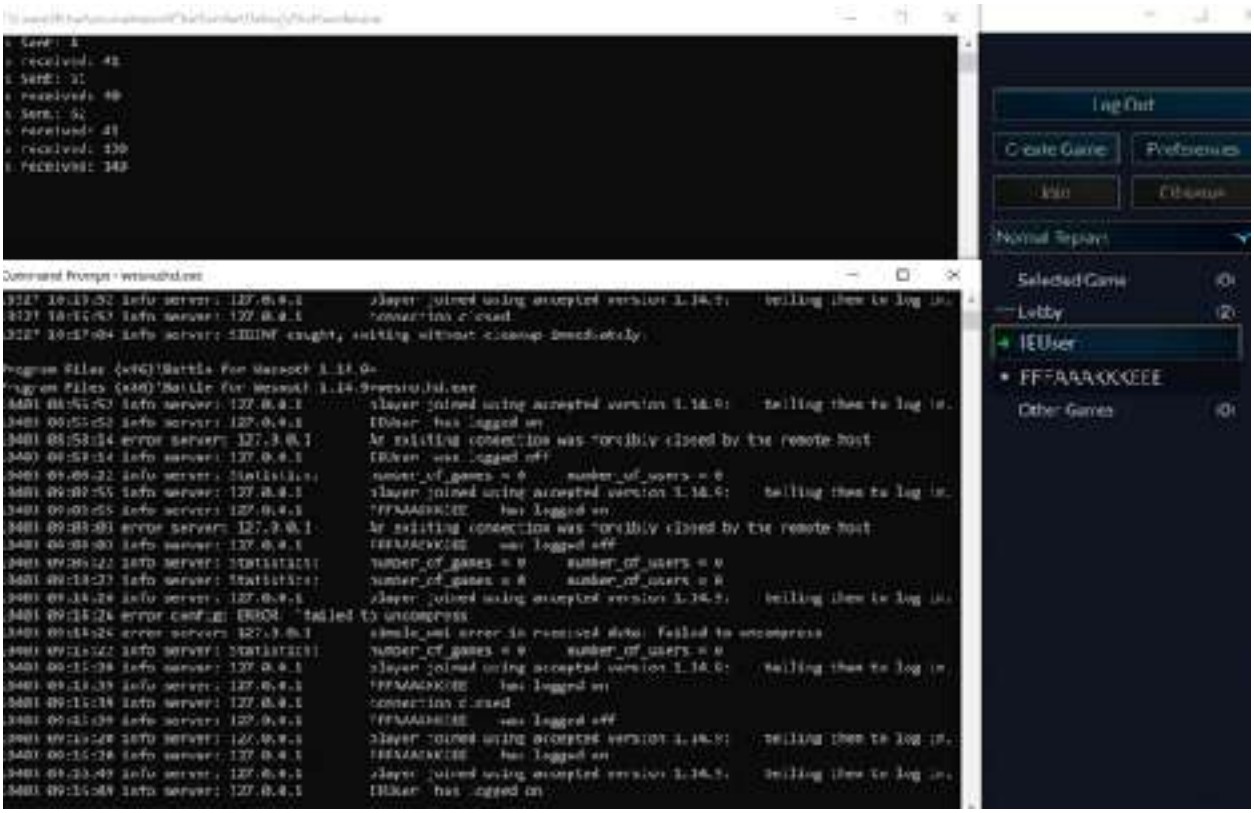
6.2 Análise de Pacotes

6.2.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

6.2.2 Identificação

Como muitos jogos, o Wesnoth tem um modo multijogador que permite que vários jogadores participem de um salão, conversem uns com os outros e joguem uns com os outros online. Nosso objetivo neste capítulo é analisar pacotes de dados usados para conectar-se ao lobby e criar um cliente que será conectado ao lobby.



6.2.3 Compreensão

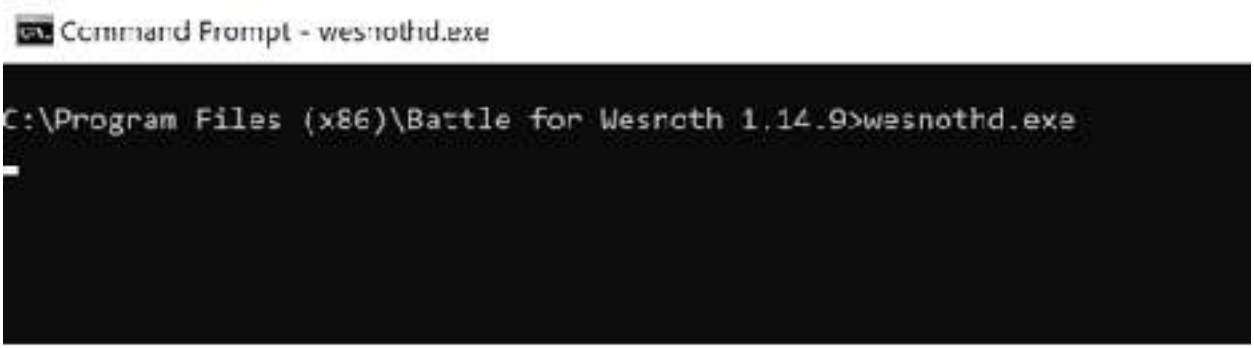
Para permitir que vários jogadores se comuniquem através da rede, todos os clientes (neste caso, executáveis de jogos Wesnoth) devem concordar com o modelo e o protocolo da rede. Eles também devem concordar com os dados que cada pacote conterá. Se houver um servidor, o servidor também deve concordar com todos esses componentes.

Como esses dados são estruturados, podemos primeiro identificar o modelo de rede e o protocolo usados no jogo. Podemos então observar os pacotes que estão sendo enviados e reverter os dados para determinar o que cada pacote está fazendo. Podemos então usar os dados nesses pacotes para criar nossos próprios clientes usando a API de socket da Windows.

6.2.4 Servidor Local

Se você iniciar o Wesnoth e clicar em jogos multijogador, verá a seguinte tela:

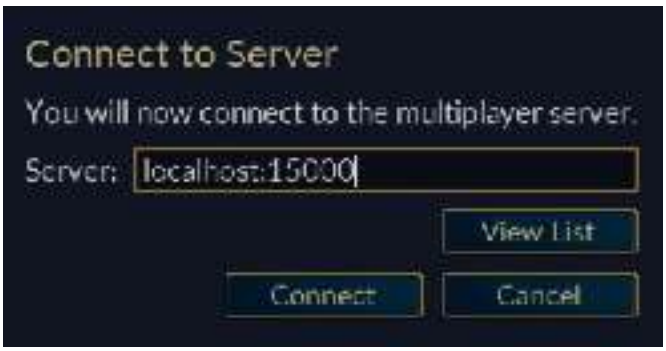
Essas entradas indicam que o Wesnoth usa um modelo cliente-servidor. Se navegarmos pela pasta de jogos Wesnoth em C:\Program Files(x86)\Battle for Wesnoth 1.14.9, você encontrará um programa chamado wesnothd.exe. Leia a documentação no site do desenvolvedor e sabemos que este é um daemon do servidor que permite que você hospede o servidor. Você pode executá-lo chamando-o do prompt de comando:



Com o servidor em execução, agora podemos nos conectar a ele a partir do Wesnoth. No capítulo anterior, discutimos como os clientes precisam saber duas mensagens para se conectar a outro host: endereço IP e porta. Neste caso, o servidor é executado em nossa máquina local. Existem várias faixas de endereço IP retidas que nunca serão usadas para alocação normal de rede. Um deles é de 127,0,0 a 127.255.255.2555, isso é reservado para o endereço loop no computador local. O componente loop-back indica que redes externas não poderão acessar essas faixas IP.

Em todos os sistemas operacionais, 127.0.1 sempre apontará para o seu host atual. Além disso, localhost é um nome de host que aponta para 127,0.0,1. Portanto, sabemos que o IP deste servidor é 127.0.0.1 ou localhost. A partir da documentação, sabemos que o servidor é executado na porta 15000 por padrão. Com essas duas informações, podemos nos conectar ao servidor.

Selecione Connect to Server e digite em localhost: 15000:



Quando você clicar em Conexão, seu cliente se juntará a um lobby multijogador. Se você observar o servidor executando sob a prompt de comando, você deve ver que ele imprimiu um evento de conexão:

```
C:\Program Files (x86)\Battle for Wesnoth 1.14.9>
C:\Program Files (x86)\Battle for Wesnoth 1.14.9>wesnothd.exe
20210403 08:55:52 info server: 127.0.0.1        player joined using accepted version 1.14.9:    telling them t
20210403 08:55:52 info server: 127.0.0.1        IFUser  has logged on
20210403 08:58:14 error server: 127.0.0.1       An existing connection was forcibly closed by the remote host
20210403 08:58:14 info server: 127.0.0.1        IFUser  was logged off
```

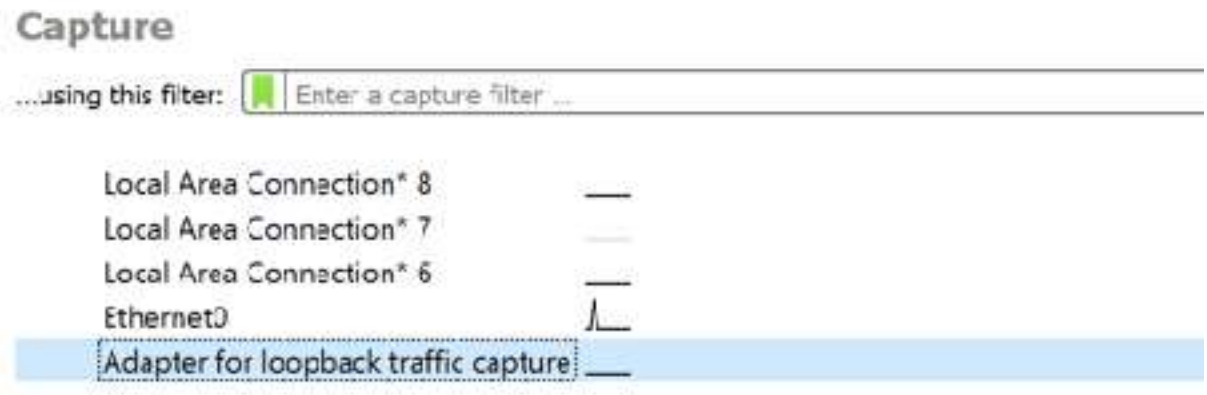
6.2.5 Pacotes de Observação

À medida que o servidor funciona, podemos desativar o cliente Wesnoth e começar a observar o processo do pacote de dados. Há muitas ferramentas disponíveis, mas para esses capítulos, usaremos o Wireshark. Isso pode ser instalado da seguinte forma:

```
choco instala wireshark
```

Ao usar o Wireshark pela primeira vez, você também precisará instalar o drive WinPcap de acordo com as instruções do programa.

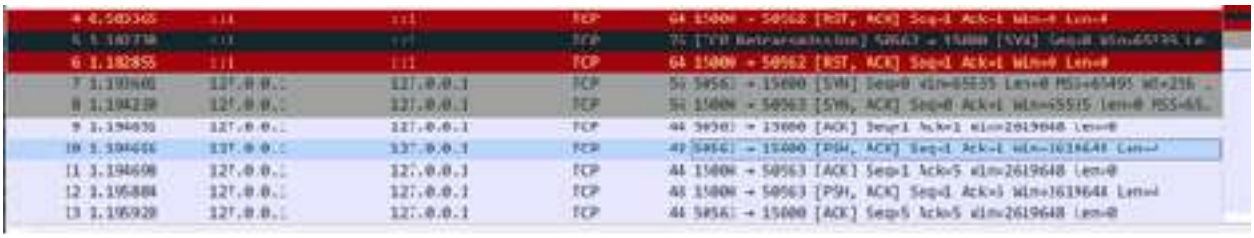
Depois de instalar o Wireshark e o driver, você pode selecionar a interface de rede que deseja observar:



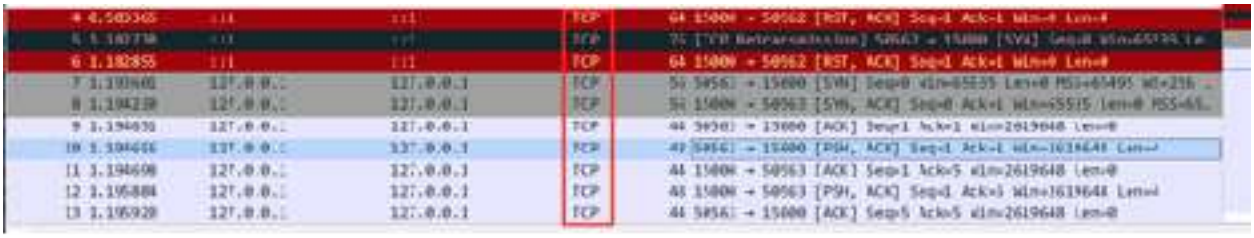
Cada interface de rede listada representa um software ou hardware conectado a uma rede pública ou privada. Por exemplo, a interface Ethernet0 listada é padrão

Uma placa de rede usada para esta máquina virtual específica se comunicar pela Internet. Essas interfaces podem variar dependendo da sua máquina virtual e do seu computador.

Neste caso, sabemos que todo o tráfego do jogo fluirá na interface loop, então selecione um adaptador para capturar o tráfego loop. Após selecionar esta opção, o Wireshark começará a monitorar pacotes de dados. Abra o Wesnoth e conecte-se ao servidor local. Ao se conectar, você deve ver o Wireshark registrando vários pacotes de dados:



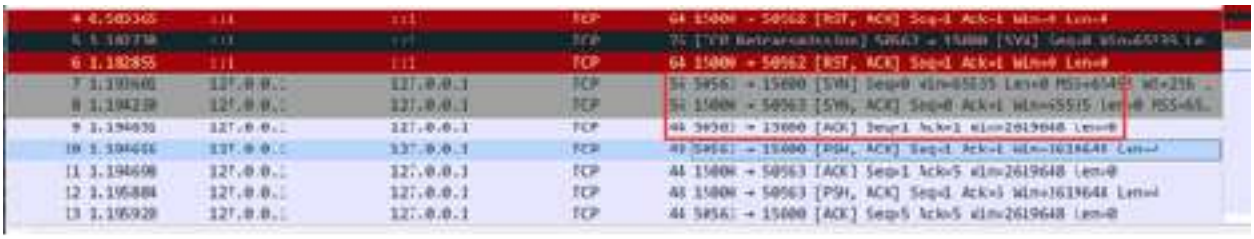
Inicialmente, isso pode parecer imparável, então vamos analisar o que realmente vemos aqui e determinar com o que nos importamos. Na coluna de protocolo, podemos ver que o Wesnoth usa TCP:



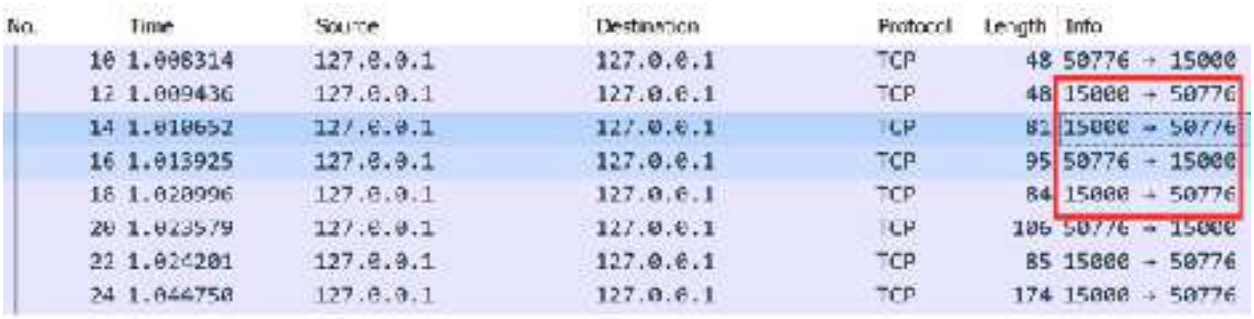
Como sabemos no capítulo anterior, o TCP inicia uma conexão frontal e a confirma ao receber o pacote de dados. Essa negociação inicial, chamada de três apertos de mão , consiste em três partes:

1. Uma das partes envia um pacote com o sinal SYN.
2. A outra parte responde com os sinais SYN e ACK.
3. A primeira parte envia o logotipo ACK.

Podemos ver esse comportamento nos primeiros pacotes destacados abaixo:

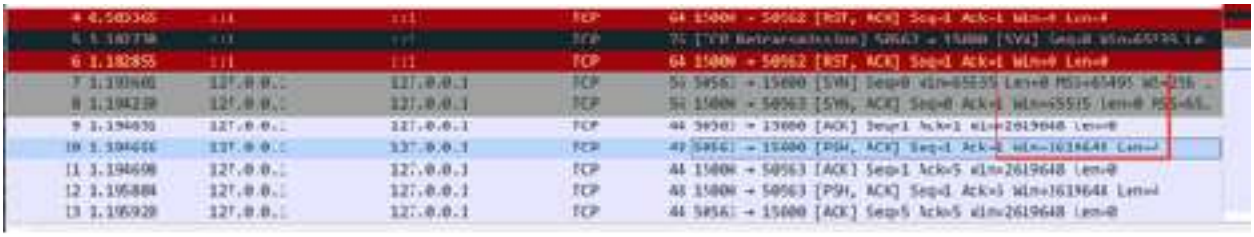


Como sabemos que 15000 é a porta do servidor, podemos confirmar que 50563 é a porta do cliente. No entanto, esse número pode não corresponder ao que você vê. Se desligarmos o Wesnoth e reiniciarmos, também veremos que esse número mudou:

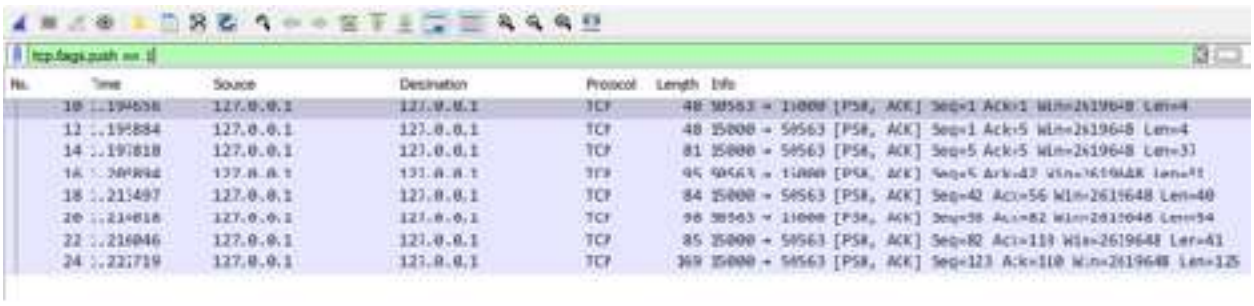


Este é um exemplo de uma porta curta ou curta. Como o cliente Wesnoth não precisa ser descoberto por outros usuários, ele pode selecionar uma porta disponível "aleatória " a cada inicialização. Quando o cliente é desativado, ele libera esta porta. Isso contrasta com os servidores, que sempre precisam ser descobertos pelo cliente na porta 15000.

Até agora, nenhum dos pacotes que verificamos contém dados. Podemos confirmar isso observando os membros do len:

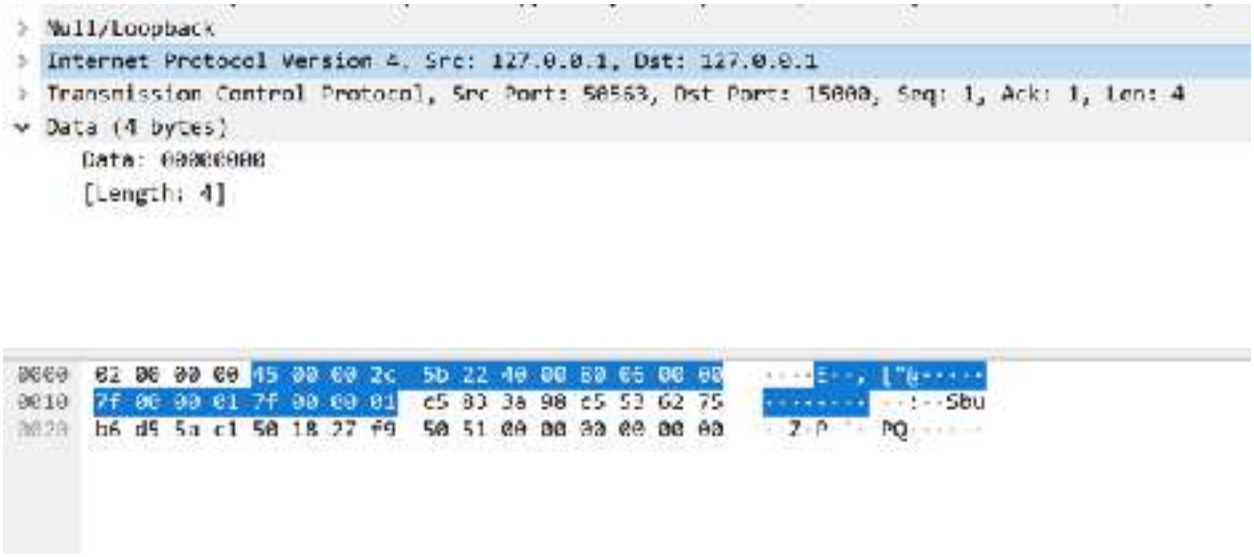


Observando todos os pacotes, podemos ver que o único pacote com dados tem a bandeira PSH. A bandeira diz à conexão TCP que enviem imediatamente quaisquer dados do pacote para o aplicativo relevante, em vez de colocá-los em um buffer. Podemos filtrar esse sinal no Wireshark e ver os pacotes com os quais nos importamos somente através do tcp.flags.push==1:

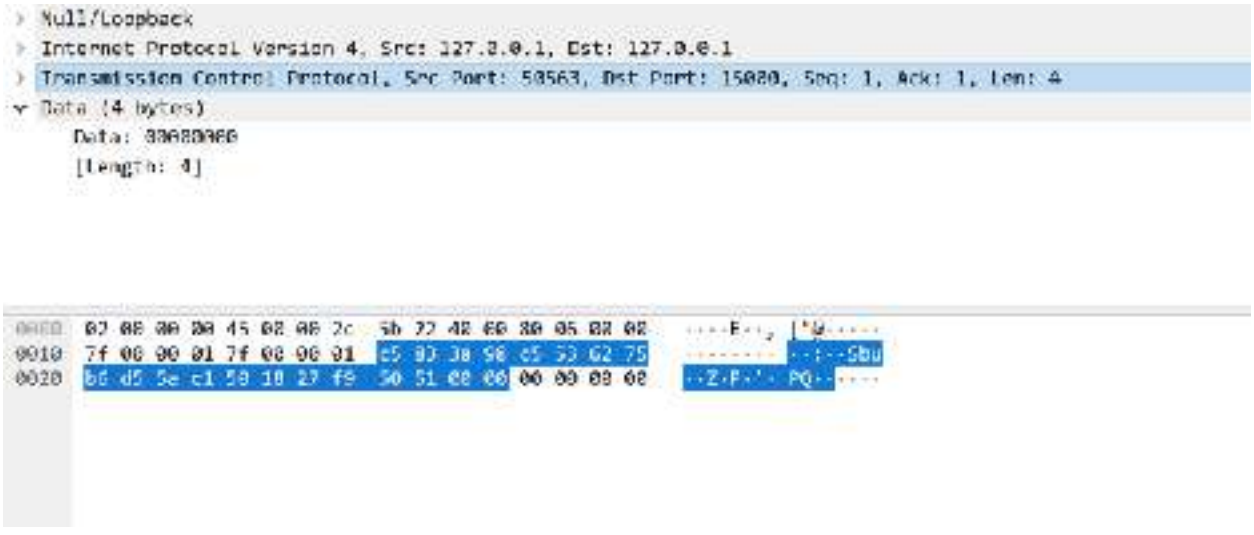


6.2.6 Estrutura de agrupamento

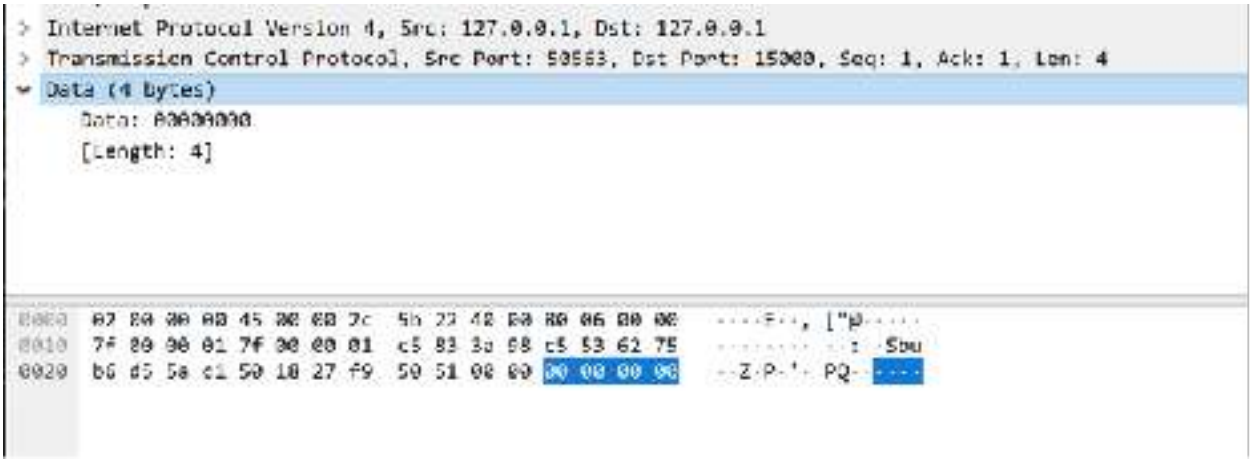
Quando você seleciona um pacote no Wireshark, ele exibe um pacote completo decomposto em segmentos diferentes. Por exemplo, se selecionarmos o primeiro pacote de dados de len 4, veremos a seguinte visão:



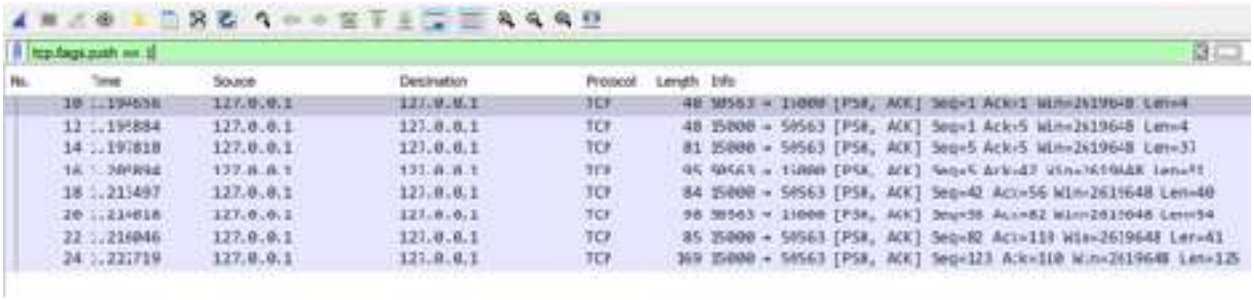
Ao selecionar diferentes componentes (como IP e TCP), as seções relevantes do pacote de dados serão destacadas na parte inferior. Por exemplo, ao selecionar o componente TCP, as seguintes seções serão destacadas:



Para fins deste capítulo, podemos ignorar os componentes IP e TCP e focar nos componentes de dados de todos os pacotes de dados:



Como os dados em pacotes de dados são frequentemente comprimidos, é difícil determinar a finalidade de um único pacote de dados isoladamente. Em vez disso, é mais fácil visualizar o fluxo geral de tráfego de rede e determinar o papel de cada pacote de dados. Em nosso exemplo, o tráfego de rede usado para conexões é o seguinte:



Sabemos que pacotes de dados 50563->15000 representam a comunicação enviada do nosso cliente ao servidor, e 15000->50563 representam a comunicação enviada do servidor ao nosso cliente. Portanto, o tráfego de rede é o seguinte:

1. Cliente ->Pacote de servidor 1
2. Servidor->Cliente
3. Cliente -> Pacote de servidor 2
4. Servidor->Cliente
5. Cliente -> Pacote de servidor 3

Como estamos escrevendo um cliente, precisamos apenas reverter os três pacotes de dados enviados do cliente para o servidor.

6.2.7 Tomada

Cada sistema operacional tem seu próprio conjunto de APIs que permite que você interaja com pilhas de rede. No Windows, essa API é chamada de Winsock. A Microsoft tem uma documentação abrangente sobre como usar esta API, incluindo o processo de criação de sockets. Aqui está isso.

A Microsoft também fornece aqui um exemplo completo do uso dessas APIs. Este exemplo criará uma conexão TCP para o IP fornecido na porta 27015 e enviará um único pacote contendo dados, o que é um teste. Ele então continuará esperando por pacotes de dados do servidor. Escreveremos código com base neste exemplo.

Um bom começo é ver como o servidor responde quando simplesmente usamos o código como ele é. Como estamos mirando um IP específico, removeremos o seguinte código do exemplo:

```
Se (argc!=2){
printf ("Uso: %s Server-Nome\n", argv[0]);
Retorno a 1;
}
```

Em seguida, podemos modificar a função getaddrinfo para usar o valor do servidor:

```
iResult=getaddrinfo ("127,0,0,1", "15000", &hints, &result);
```

Use essas alterações para compilar o código e executar arquivos executáveis. Se o Wireshark ainda estiver registrando logs, você deve notar que seu host enviou um pacote contendo dados para o servidor, o que é um teste. Se você verificar o processo do servidor, verá a seguinte mensagem:

```
20210403 09:34:08 error server: 127.0.0.1       incorrect handshake
```

6.2.8 Agrupamento Inverso

Esta mensagem instrui que o primeiro pacote enviado pelo nosso cliente deve iniciar algum tipo de aperto de mão. Vamos comparar se a primeira mensagem registrada pelo servidor é uma conexão válida:

Voltando à nossa captura Wireshark, podemos ver que o primeiro pacote enviado pelo cliente contém 000000 00. O servidor então responde com dados. A partir disso, podemos presumir que os dados 0000000000 são interpretados pelo servidor Wesnoth como o início de um aperto de mão.

Podemos modificar nosso exemplo de socket para executar esse comportamento. Primeiro, exclua o seguinte código porque vamos escrever nosso próprio código de envio:

```
//Envie buffer inicial
Iresuult = send(conexão, Sendbuff, (int)streng (sendbuf), 0); Se(
iResult==SOCKET_ERROR){
printf ("Falha no envio, erro %d\n", WSAGetLastError()); Closescocket
(conectividade);
wsaacleanup ();
Retorno a 1;
}

printf ("Bytes enviados: %ld\n", iResult);

//Desligue a conexão porque os dados não são mais enviados.
iResect=shutdown (conexão, SD_SEND);
Se(iResult==SOCKET_ERROR){
printf ("Falha de desligamento, erro: %d\n", wsagetlasterrol ());
Closescocket (conectividade);
wsaacleanup ();
Retorno a 1;
}
```

Em seguida, crie um buffer para salvar nossos dados 00000000:

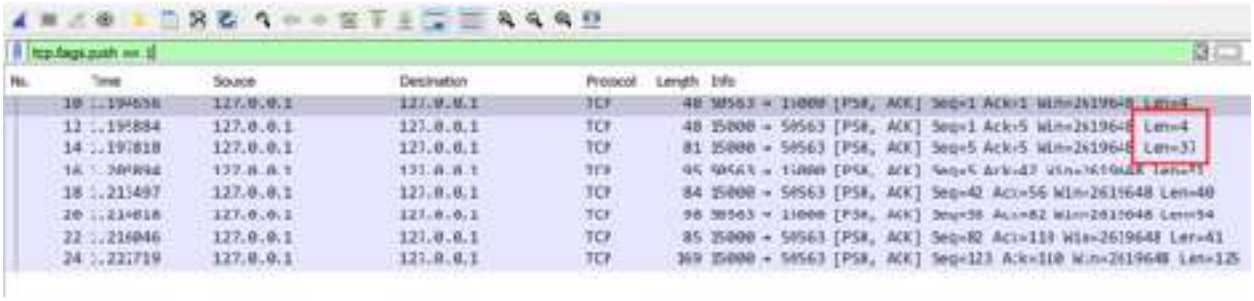
```
const unsigned char buff_handshake_p1[]={0x
00,0x00,0x00,0x00
};
```

Por fim, adicione o seguinte código para enviar esses dados e receber um único pacote de dados retorno:

```
iResult=send(Conect, (const char*)buff_handshake_p1, (int)
sizeof(buff_handshake_p1), 0);
printf ("Bytes enviados: %ld\n", iResult);
```

```
Iresuult = recv (conect, recvbuff, recvbuff, 0); printf ("
Bytes recebidos: %d\n", iResult);
```

Se você executar este programa, receberá 41 bytes. Isso equivale a duas respostas enviadas pelo servidor em Wireshark, indicando que o primeiro pacote enviado pelo cliente inicia um aperto de mão:



Na mensagem do servidor, podemos ver que o próximo pacote de dados pelo qual o cliente é responsável está enviando sua versão atual. O exemplo de dados deste pacote de dados é o seguinte:

```
0x00,0x00,0x00,0x2f, 0x1f, 0x8b, 0x0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0x2b, 0x2b, 0
x2b, 0x2a, 0x8b, 0x00,0x00,0xxx0x0x0x0x0x0d2,5, c4,5, c4,5, c4,5, c4,5, c4,5, c4,5, c4,5, c4,5, c4,5, c
4,5, c4,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,c,c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,
5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,c,5, c,5, c,5, c,5, c,5, c,5
, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c
,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5,
```

Mesmo que seja convertido para ASCII, nossa versão do jogo (1.14.9) não aparecerá nesses dados. Isso ocorre porque, como na maioria dos jogos, o Wesnoth comprime todos os dados por padrão. Em capítulos posteriores, examinaremos os esquemas de compressão usados para que possamos criar pacotes com dados personalizados. Entretanto, neste capítulo, não precisamos fazer isso porque esses dados não mudam. Você pode verificar isso adicionando ao mesmo servidor várias vezes enquanto o Wireshark está em execução.

Vamos adicionar este pacote ao nosso programa para envio:

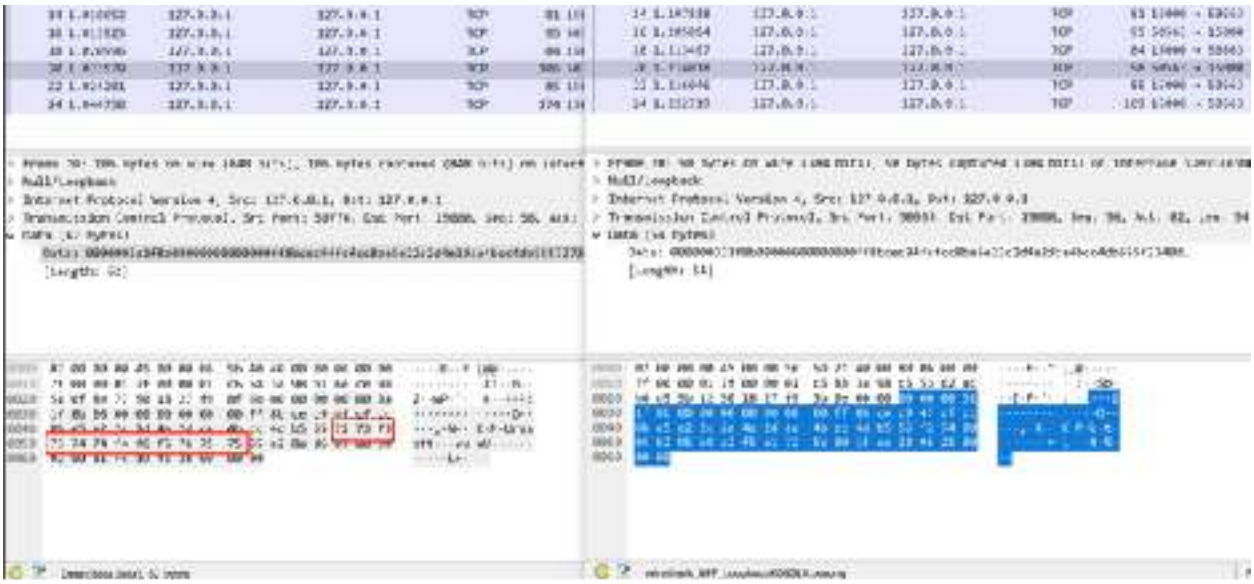
```
const unsigned char buff_handshake_p2[]={
0x00, 0x00, 0x00, 0x2f, 0X01F, 0x8b, 0X008, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x
8b, 0x2e, 0x4b, 0X00000CE, 0X000CE, 0X000CE, 0X00CE, 0XXXXXXXXX, DXXXXXXXXXXX0X0X0X0X
0X0F95, F95, F95, F95, F95, F95, 6C, F95, 6C, 6C, 38.8, F95, 5C, 5C, 5C, 5C, 38C, 38C, 5C,
5C, 5C, 5C, 38C, 5C, 5C, 5C, 5C, 5C, 5C, 5C, 5C, 38C, 5C, 5C, 38C, 5C, 5C, 5C, 38C, 38C, 5C,
38C, 5C, 38C, B0CC, 5C, 5C, B0CC, B0CC, B0CC, 38CC, 5C, B0CC, B0CC, 5CC, 38CC, B0CC, 5CC
```

```
0x00
};

iResult=send(Conect, (const char*)buff_handshake_p2, (int)
sizeof(buff_handshake_p2), 0);
printf ("Bytes enviados: %ld\n", iResult);

Iresuult = recv (conect, recvbuff, recvbuff, 0); printf ("
Bytes recebidos: %d\n", iResult);
```

Com este pacote adicional, o servidor Wesnoth agora considerará que o cliente enviará a versão do jogo para eles antes de fechar a conexão:

```
info server: 127.0.0.1        player joined using accepted version 1.14.9:    telling them to log in.
info server: 127.0.0.1        connection closed
```

Por fim, podemos adicionar o nome enviado pelo cliente. Como podemos controlar esse campo, podemos usá-lo para observar o esquema de compressão que está sendo usado . Ao executar o Wireshark, conecte-se ao servidor usando dois nomes de usuário (um curto e um longo). Em nomes de usuário longos, certifique-se de repetir vários caracteres em uma linha. Isso nos permitirá detectar o modo. Neste capítulo, usaremos exemplos de FfaakkKeee e IEUser. Seus pacotes de dados relacionados são os seguintes:



A área destacada provavelmente representa o nome compactado do usuário. Vamos tentar enviar dados do Ffaakkeee, mas modificar ligeiramente os bytes do nome:

```
const unsigned char Buff_Send_Nome[]={
0x00, 0x00, 0x00, 0x3a, 0x1f, 0X08B, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x
8b, 0xce, 0xc9, 0x4f, 0X000008B, 0X08B, 0X08B, 000008B, 0000074, F74, 4, 4, F74,
4, 4, 4, F74, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,

};

iResult=send(conexão, (conexão car*)buff_Send_Nome, (int)
sizeof(buff_Send_Nome), 0);
printf ("Bytes enviados: %ld\n", iResult);
```

Se olharmos para o servidor, veremos o seguinte erro:

```
info server: 127.0.0.1        player joined using accepted version 1.14.9:    telling them t
error config: ERROR: 'failed to uncompress'
error server: 127.0.0.1       simple_wml error in received data: failed to uncompress
```

Isso verifica que o pacote está sendo comprimido e até indica um esquema de compressão (simple_wml). Quando quisermos criar nossos próprios pacotes de dados, podemos usar essas informações em capítulos subsequentes. Para este capítulo, podemos modificar apenas o buff_send_nome para incluir os dados originais:

```
const unsigned char Buff_Send_Nome[]={
0x00, 0x00, 0x00, 0x3a, 0x1f, 0X08B, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0x8b, 0xce, 0xc9, 0x4f, 0X00008B, 0X08B, 0X08B, 0X0008B, 0X074, F75, 4, F76, 4
, 4, A, 4, 4, 4, 4, 4, 4, 4, B, B, 4, 4, 4, 4, 4, 4, B, B, 4, 4, B, B, 4, B, 4, 4, B, B,
4, 4, 4, B, 4, 4, 4, 4, 4, 4, B, 4, B, 4, B, B, 4, 4, 4, 4, 4, B, B, 4, 4, 4, 4, 4, 4, 4
, 4, B, 4, 4, 4, B, B, 4, B, 4, 4, 4, 4, 4, B, B, B, 4, 4, B, 4, 4, B, B, B, B, 4, 4, 4,
4, B, 4, 4, 4, 4, 4, 4, 4, B, 4, 4, 4, 4, 4, B, 4, B, 5, 4, 4, 4, 4, 4, 4, B, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, B, 4, 5, 4, B, 4, 4, 5, B, 5, 5, 4, 4, B, 4, 5, 5, 4, B, 4, 5, 4, B, 4,

};
```

Com essa mudança, nossos clientes agora se conectarão ao servidor usando o nome FFFAAKKKeee. Se você usar um cliente legal para entrar no lobby, notará que nosso cliente também está conectado.

O código completo deste cliente pode ser encontrado no Apêndice A.

6.3 Ré
pacote de dados

6.3.1 Objetivos

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

6.3.2 Identificação

No capítulo anterior, identificamos os pacotes de dados usados para se conectar ao servidor Wesnoth. Então escrevemos um cliente para reproduzir esses pacotes. Neste capítulo, identificaremos pacotes de dados usados para enviar mensagens de bate-papo e reverteremos sua estrutura. Isso nos permitirá criar nossos próprios pacotes legítimos, não apenas reproduzi-los.

6.3.3 Compreensão

Para permitir que o cliente e o servidor se comuniquem pela rede, ambas as partes devem concordar sobre como construir os dados em cada pacote de dados. Como essa estrutura deve ser reversível para cada lado, também podemos revertê-la. Quando os dados são revertidos, podemos modificá-los e fazer algo oposto ao processo reverso para criar um novo pacote de dados.

6.3.4 Pacote de bate-papo

Semelhante aos executáveis reversos, é útil ter um contexto ao reverter pacotes de dados. Para arquivos executáveis, esse contexto geralmente é a string que observamos em arquivos executáveis. Quando se trata de pacotes de dados, esse contexto é um certo tipo de dados que podemos controlar no pacote de dados.

Normalmente, você controla vários dados em um pacote de dados. Por exemplo, a maioria dos jogos permite que você defina seu nome. Outros jogos permitem que você se conecte a várias versões ou certos mods. Ambas as partes podem fazer você pensar em algumas

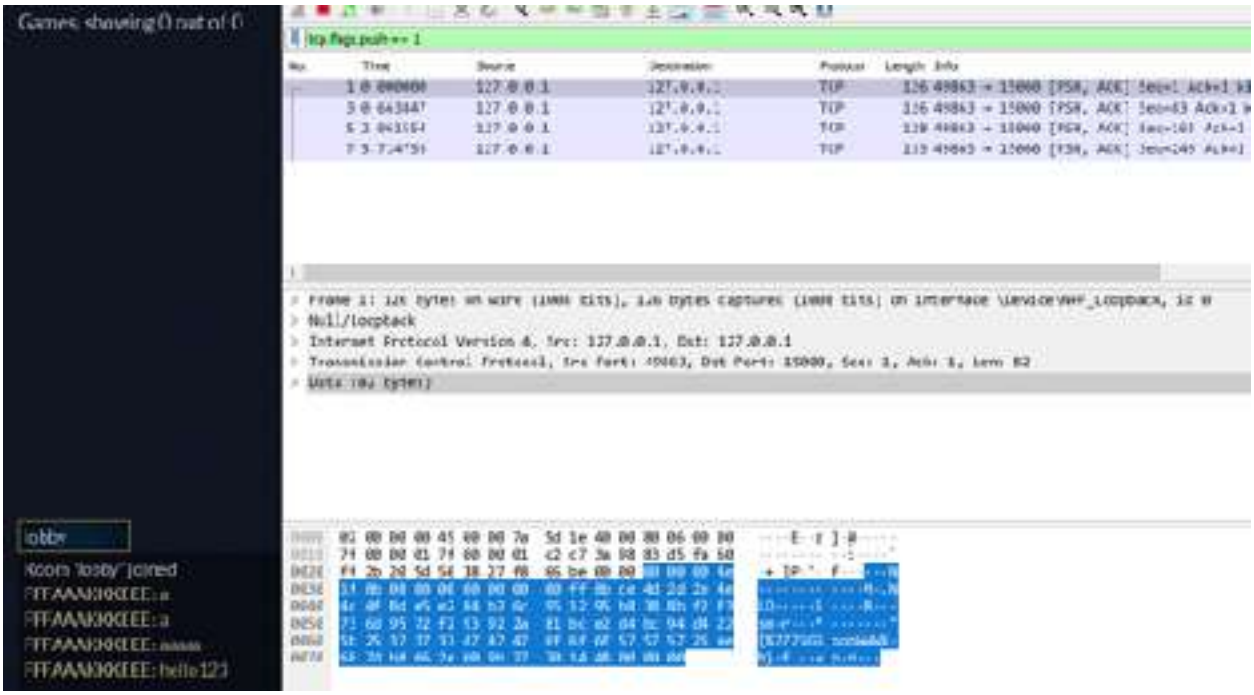
Dados com certos pacotes de dados. No entanto, um dos dados mais fáceis de controlar são as mensagens de bate-papo do jogo. Como o Wesnoth permite que os jogadores enviem mensagens de bate-papo, este é o contexto que usaremos neste capítulo.

Inicie o Wesnoth e conecte-se ao seu servidor local usando um usuário chamado FFFAAKKeE, como no capítulo anterior. Após a conexão, inicie o Wireshark para registrar pacotes de dados. Para ajudar a reverter o pacote de dados do jogo, gostaríamos de responder a algumas perguntas:

1. Quaisquer elementos aleatórios ou temporais são codificados no pacote de dados?
2. Podemos observar o padrão de letras na bolsa?
3. Podemos observar quaisquer caracteres legíveis humanos em pacotes de dados?

Para responder a essas perguntas, podemos enviar as quatro mensagens de bate-papo a seguir:

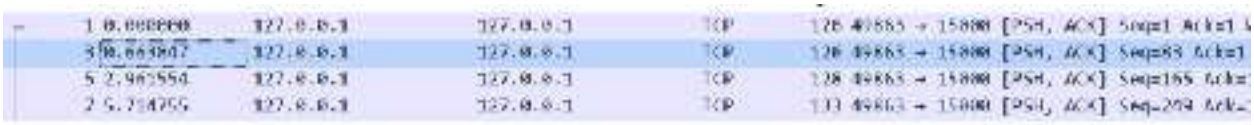
- *a*
- *a*
- *Ahhh*
- *Olá 123*

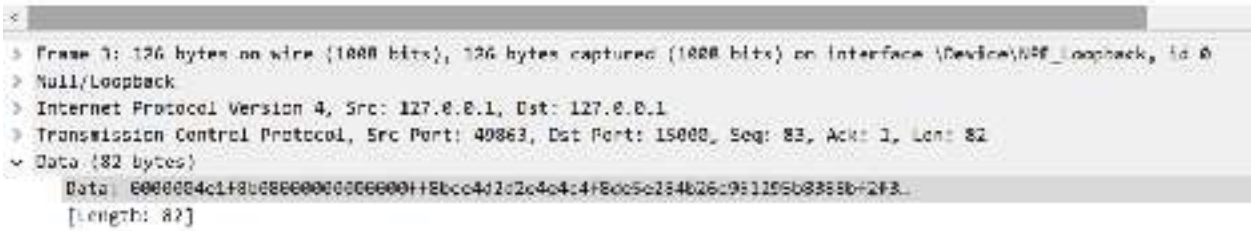


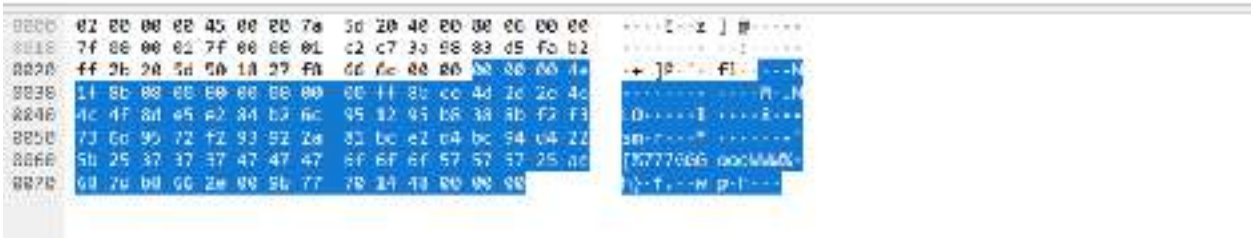
As duas primeiras mensagens são iguais, o que nos ajudará a determinar se a mesma mensagem resultará no mesmo pacote de dados. A terceira mensagem é repetida várias vezes, permitindo-nos observar qualquer tipo de padrão de dados. Por fim, a última mensagem nos diz imediatamente se ela é legível no pacote de dados.

6.3.5 Casos de Teste

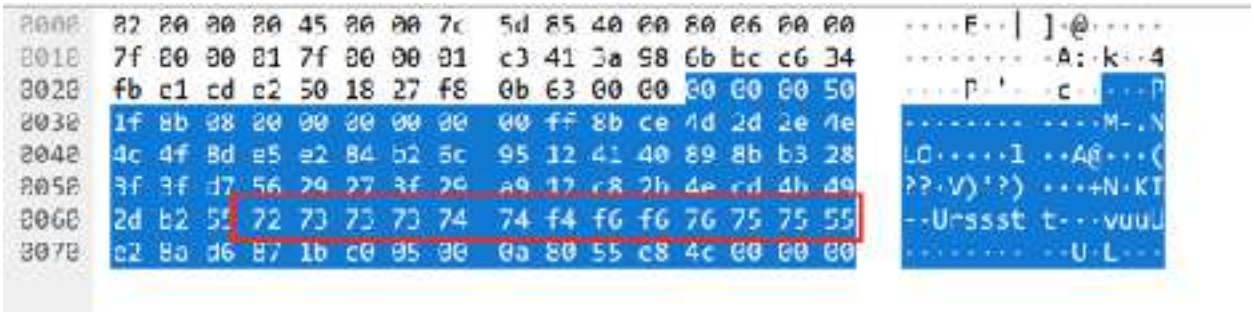
Ao verificar os dados dos dois primeiros grupos (A e A), podemos ver que seus dados são os mesmos. Isso significa que os pacotes de dados não contêm carimbos de data ou quaisquer outros fatores exclusivos:







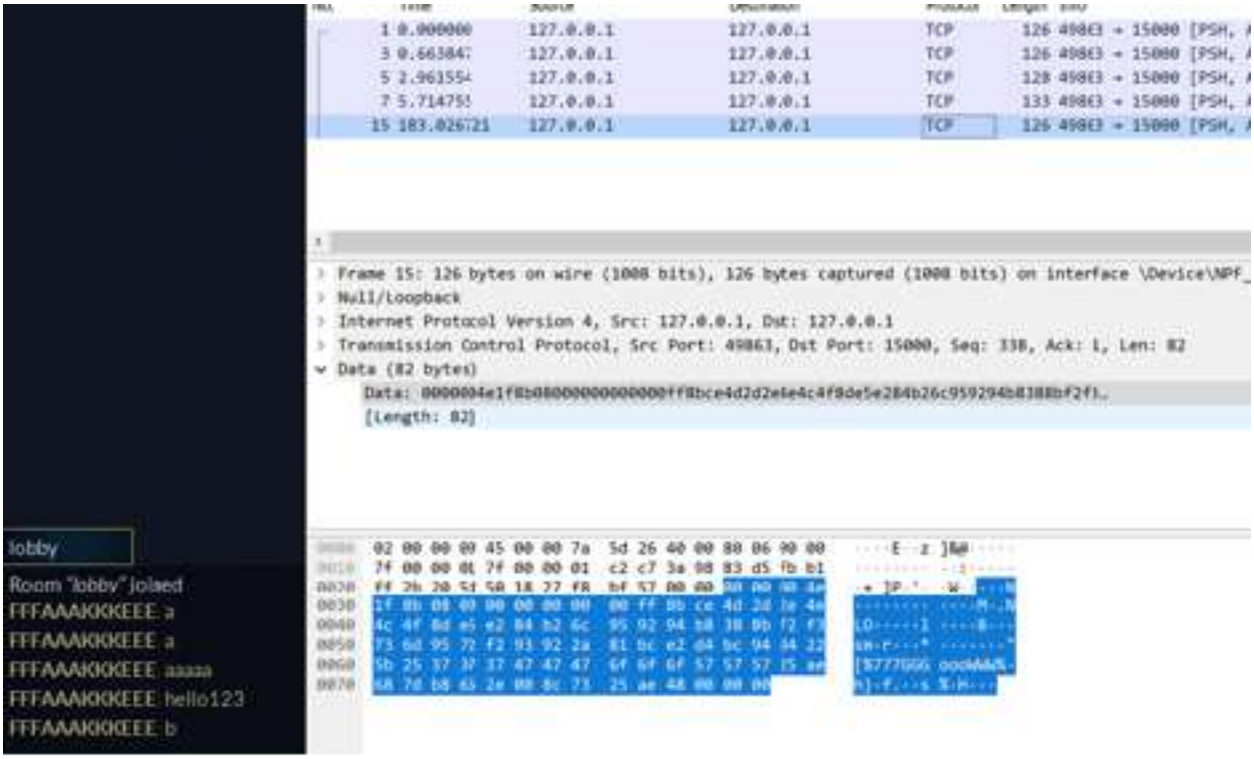
Em seguida, verificamos os dados do terceiro pacote (aaaaa) e vemos um padrão próximo ao fim:



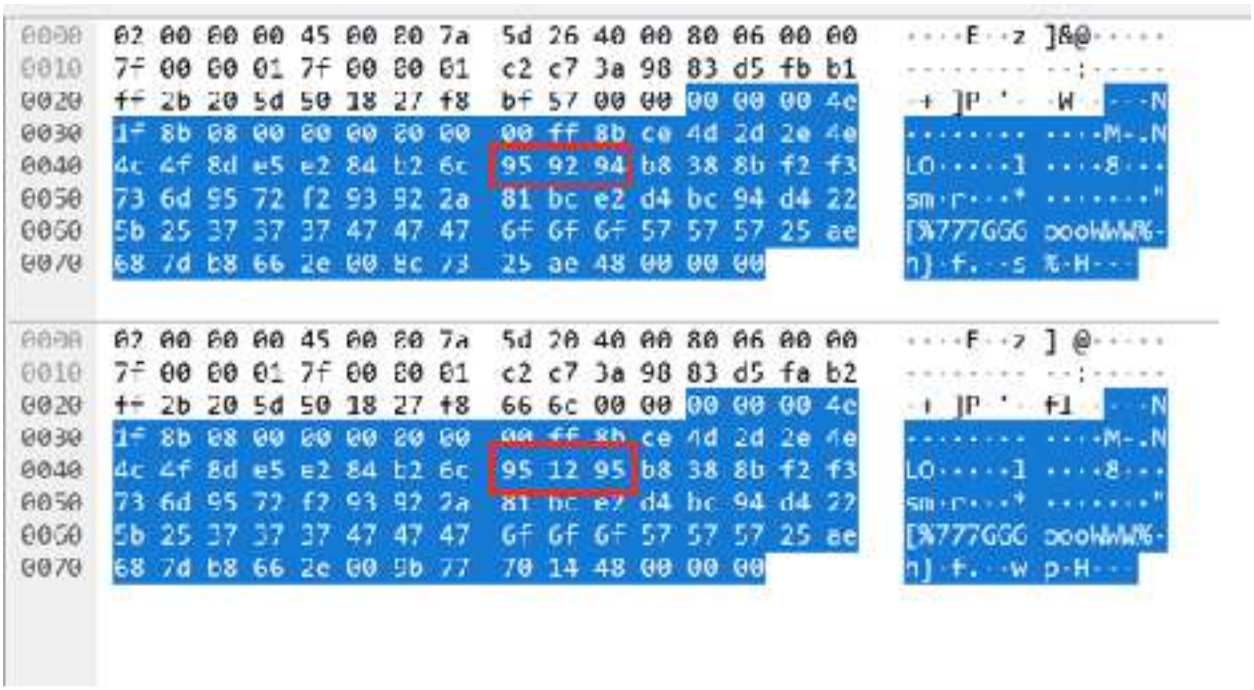
Esse modo é muito longo para a AAAA, e sua estrutura 3-3-3 está mais próxima do nome do nosso jogador (FFFAAKKeee). Podemos ver que esse padrão também aparece em todas as nossas mensagens, indicando que um elemento deste pacote deve conter o nome do nosso jogador.

Por fim, verificando os dados do último pacote, não podemos observar nada semelhante ao hello123.

Em seguida, vamos observar as diferenças entre os pacotes da mensagem A e da nova mensagem b para ajudar a determinar como processar um único caractere:



Comparando os dois pacotes lado a lado, descobrimos que a modificação de um único caractere resultou na alteração de 2 bytes no meio do pacote e na alteração de vários bytes no final:

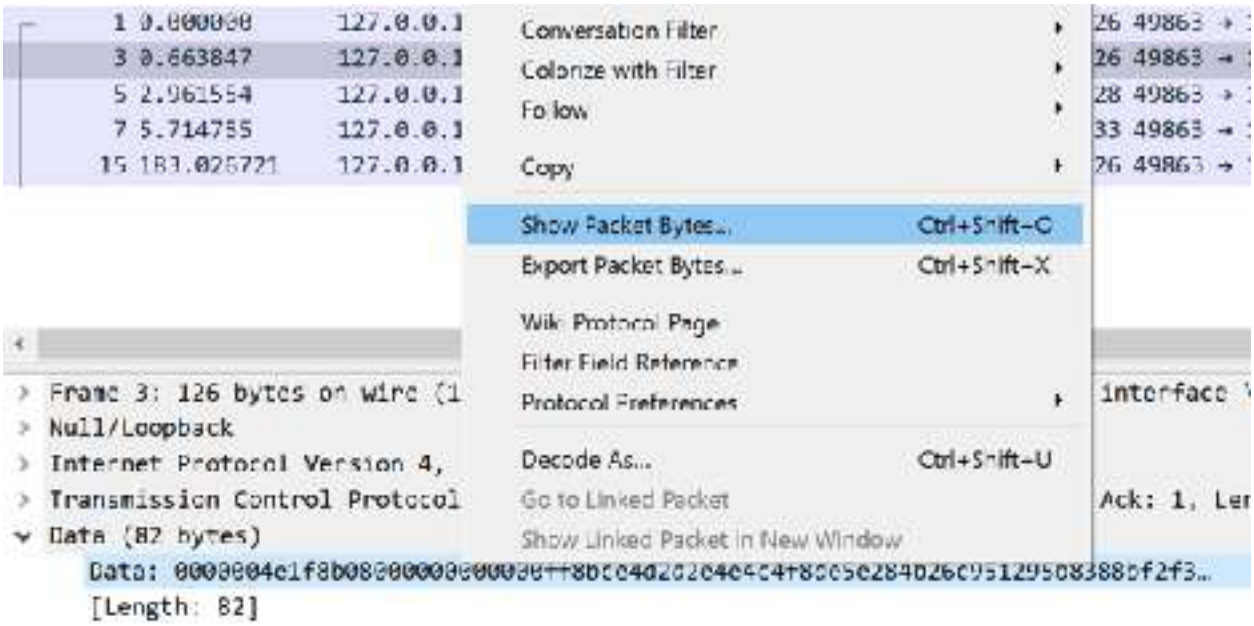


Isso sugere que nosso texto não está sendo mapeado para pacotes individualmente e que processamento adicional está em andamento. Com essas informações, podemos desligar o Wesnoth e parar de registrar pacotes em Wireshark. No entanto, certifique-se de manter o Wireshark em execução para que possamos capturar os dados do pacote enquanto analisamos mais detalhadamente.

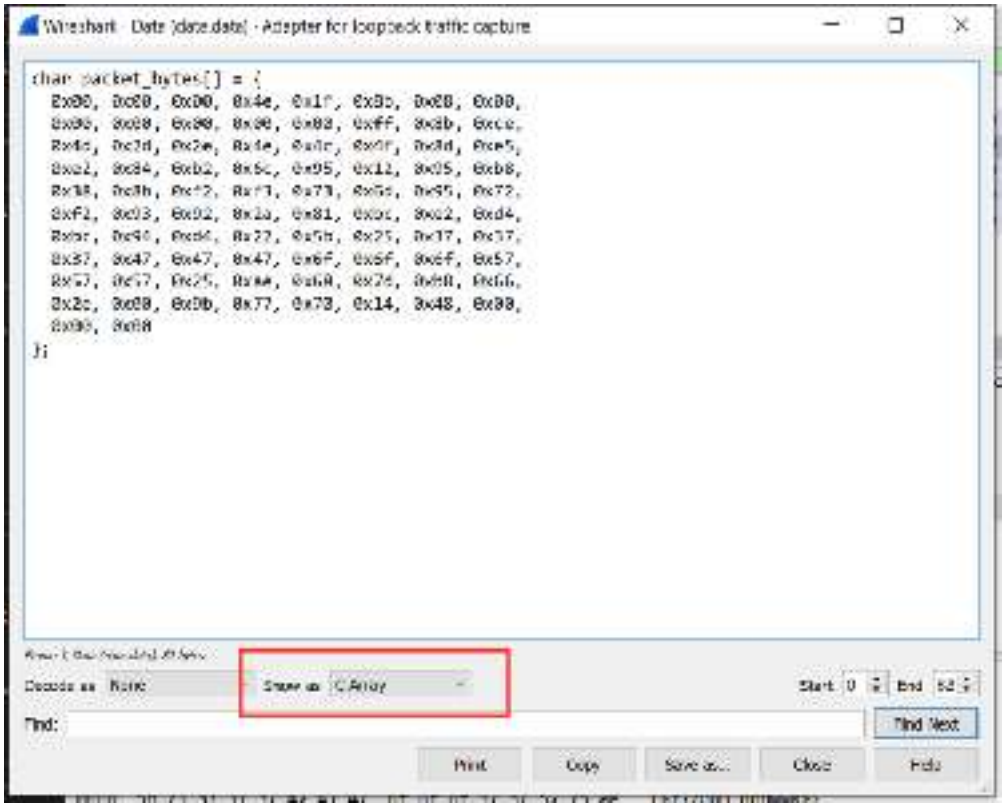
6.3.6 Modificação do pacote de dados

Agora, vamos dar uma olhada em como o servidor responde se modificarmos um pacote. No capítulo anterior, escrevemos um cliente que se conectará ao servidor usando o nome de usuário Ffaakkeee. Podemos estender esse código e enviar a mensagem de bate-papo A após a conexão.

Em Wireshark, clique no primeiro ou segundo pacote (a mensagem) e selecione os dados. Clique com o botão direito e selecione "Exibir bytes de pacote":



Na janela que aparece, selecione "Array C" na seção "Exibir como". Isso formatará os dados do pacote em um array que pode ser colocado diretamente no seu código.


Wireshark · Data (data.data) · Adapter for loopback traffic capture
char packet_bytes[] = {
0x00, 0x00, 0x00, 0x4c, 0x1f, 0x8b, 0x08, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x8b, 0xc0,
0x4d, 0x20, 0xc4, 0xc4, 0xf8, 0xc0, 0x5e, 0x28,
0x40, 0x26, 0xc9, 0x51, 0x29, 0x50, 0x83, 0x88,
0x5f, 0x2f, 0x33, 0x0a, 0x0a, 0x0a, 0x0a, 0x0a,
0x0a, 0x0a, 0x0a

Depois que enviamos o pacote de aperto de mão, versão e nome de usuário, podemos adicionar o seguinte código para enviar a mensagem de bate-papo:

```
const unsigned char packet_bytes[]={
0x00, 0x00, 0x00, 0x4e, 0x1f, 0x8b, 0x008, 0x00, 0x00, 0x00,
0x00, 0x00, 0x8b, 0xce, 0x4d, 0x2e, 0x4e, 0x0008C, 0x008B, 0x
08B, 547, x0B, 5A, 547, X0B, 5A, 547, 5B, 5B, 5a, 5B, 5B, 5a, 5
B, 5a, 5B, 547, 5B, 5a, 5B, 5a, 5a, 5a, 547, 5B, 5a, 5B, 5a, 5B,
5a, 547, 5B, 547, 5B, 5a, 5B, 5a, 5a, 5B, 5a, 5B, 547, 5B, 547,
5B, 5B, 5B, 5B, 547, 5B, 5a, 5B, 5B, 5B, 5a, 5B, 5a, 5B, 5B, 5a,
5B, 5a, 5B, a, 5B, 547, 5a, 547, 5B, 5a, 5B, 5B, 5B, a, 5B, 5B, a
, 38B, a, a, a, 5B, a, a, a, a, a, 547, 547, 547, a, 5B, 5B, a, 5B
, a, 547, a, a, 5B, 38B, 547, 38B, a, 5B, a, a, 547, a, a, a, a,
547, a, a, a, a, a, 547, 5B, 547, 5B, 5B, a, a, a, a, a, a,
0x00,0x00
};

Iresuult=send(conexão, (conexão) packet_bytes, (int)
sizeof(packet_bytes), 0);
printf ("Bytes enviados: %ld\n", iResult);
```

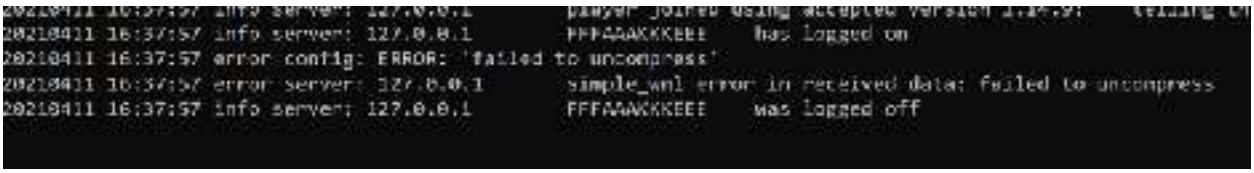
Executar este código enviará uma mensagem de bate-papo, como se estivesse conectada:

Na seção acima, vemos que a diferença entre as mensagens de bate-papo A e b é uma mudança de 2 bytes. Vamos alterar apenas os 2 bytes acima para ver como o servidor responde:

```
const unsigned char packet_bytes[]={
0x00,0x00,0x00,0x00,0x4e, 0x1f, 0x8b,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0x82,0x4d
,0x2e, 0x4d,0x4d,0x4d,0x4d,0x4d,0x4d,0x4d,0x4d,0x0x0x0d,5, cc, 0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x84,5, c
,5, c,4, c,5, c,5, c,5, c,c,5, c,5, c,5, c,5, c,c,5, c,c,5, c,c,5, c,5, c,c,5, c,c,5, c,c,4, c,c,c,c,c,5, c4, c,4, c,4
, c4, c,4, c,5, c,5, c,5, c,5, c,c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c4, C,5, c,c,5, c,c,5, C,5, c4, c4,
```

```
0x38, 0x8b, 0xf2,0xf3,0x73,0x6d, 0x95,0x72,0xf2,0x93
,0x92,0x2a, 0x81,0xB, 0xB, 0x24,0x94,0xd4,0x22,0x5b,
0x5b, 0x25, 0x37, 0x4, 0xx0xxx0x0x64,7, 25,7, 25, Xx0x
0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x5,
75,7, 75,7, 75,7, 7, 75,7, 75,7, 75,7, 75,7, 75,7, 75,7,
75,7, 75,7, 25, 25,

};
```

Usar este pacote para executar o código resultará nos seguintes erros no servidor:



Esta mensagem de erro, juntamente com como a alteração em uma única letra leva a múltiplas modificações nos bytes em um pacote, indica que pelo menos algumas partes do pacote foram comprimidas.

6.3.7 Compressão

Compressão é o processo de aquisição de dados de entrada e redução de tamanho. Uma das técnicas de compressão mais simples é combinar dados duplicados. Por exemplo, a string aaaaaaaaaaaaa pode se transformar em 10A. Ao descomprimir, o descompressor saberá estender 10A de volta para AaaaaaaAAAAA. Existem muitas maneiras de comprimir dados, e alguns formatos populares são ZIP e RAR. Assim como arquivos executáveis são sempre diferentes de outros dados (como imagens), diferentes formatos de compressão também são diferentes uns dos outros.

Wesnoth é um jogo multiplataforma que suporta Windows e Linux. Portanto, sabemos que, independentemente do formato usado, ele deve ser executado em ambos os sistemas operacionais. Em sistemas baseados em Linux, os dois formatos de compressão mais populares são gzip e bzip2. Iniciaremos nossa investigação com esses formatos.

O prompt de comando padrão do Windows não oferece bom suporte para operações de dados. Para nos ajudar com nossa investigação, usaremos outro simulador de terminal chamado Cmder. Também instalaremos o pacote gzip. Ambos podem ser instalados em Powershell usando Chocolate:

```
choco Instalação cmder-y
choco Instalação gzip-y
```

Para testar ambas as técnicas de compressão, crie um arquivo de texto chamado test.txt. Neste arquivo, adicione uma linha de texto. Em seguida, abra Cmder (C:\Tools\Cmder.exe) e navegue para o diretório que contém este arquivo de texto. Executar o seguinte comando para criar a versão gzip do arquivo:

```
Teste gzip.txt
```

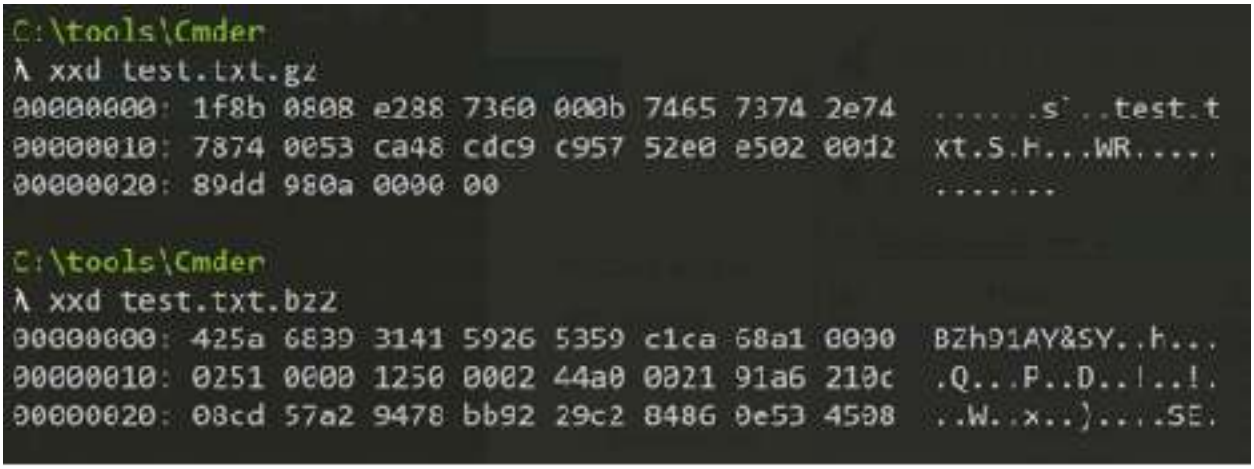
Por padrão, o gzip excluirá o arquivo original. Recrie o test.txt da mesma maneira para que você possa criar uma versão bzip:

```
Teste bzip2.txt
```

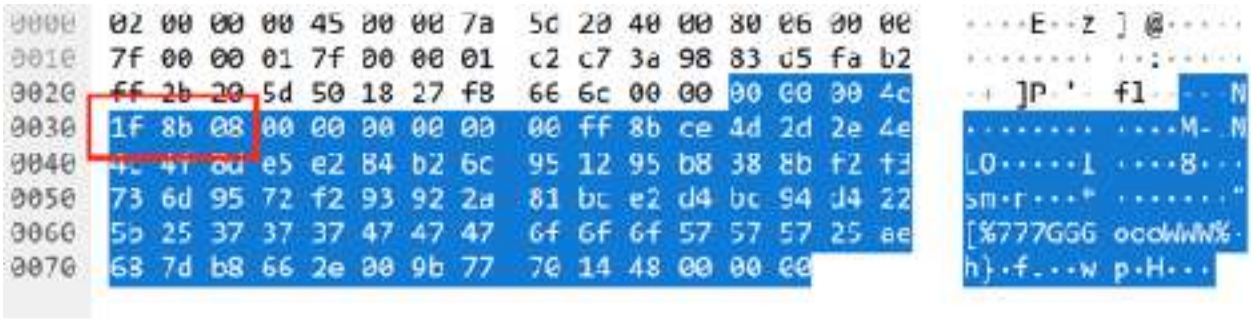
Agora deveria haver dois arquivos: test.txt.gz e test.txt.bz 2. Em seguida, vamos verificar como são os bytes desses arquivos. Para isso, podemos usar uma ferramenta chamada xxd:

```
xxd test.txt.gz
xxd test.txt.bz 2
```

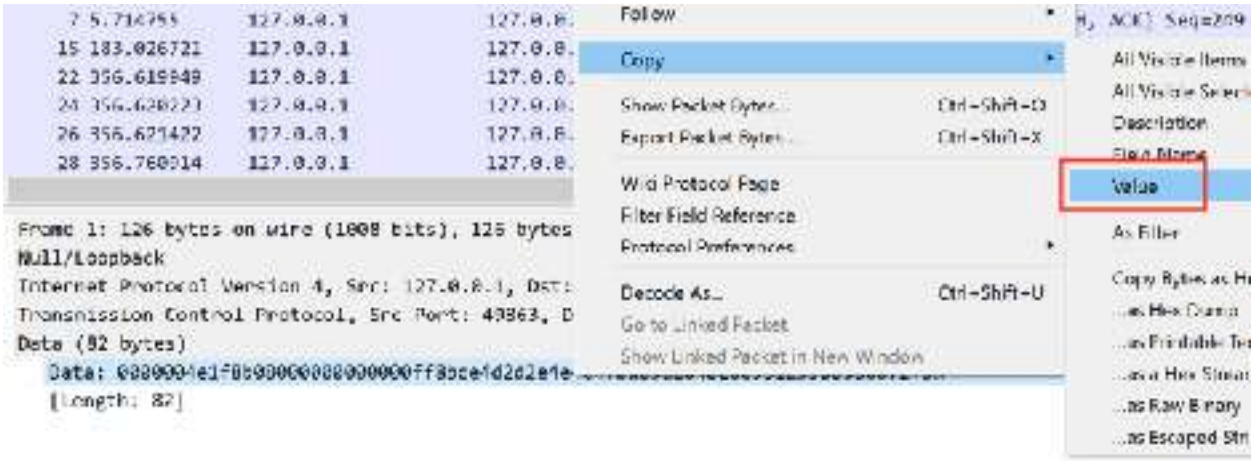
Seus resultados devem ser semelhantes aos seguintes:



Ao comparar esses pacotes com um dos pacotes, você deve notar imediatamente que o byte inicial (0x1f8b) no pacote aparece:



Isso sugere que uma parte do pacote pode ser comprimida usando gzip. Podemos verificar isso tentando descomprimir o pacote de dados real. Em Wireshark, selecione um pacote de dados, clique com o botão direito nos dados e selecione Copy -> Valor:



Podemos usar o xxd novamente para converter esses dados em arquivos gzip. Para isso, primeiro usaremos o comando echo para imprimir os dados no terminal. No entanto, passaremos o texto impresso para xxd por meio de um operador de tubo (|), em vez de apenas imprimi-lo. Podemos então usar o interruptor -r para informar xxd sobre a operação de reversão (ou criar um arquivo a partir do hexadecimal) e usar o interruptor -p para informar xxd sobre a leitura de qualquer coisa que digite, neste caso, o comando echo. Por fim, usamos o operador de redireção (>) para salvá-lo em um arquivo compactado:

```
Eco "1f8b08..." |xxd-r-p->file.gz
```

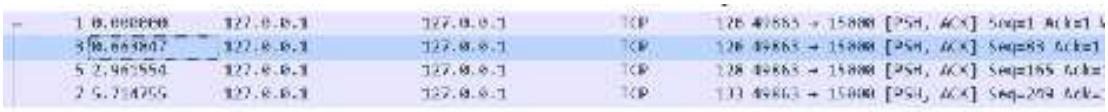
Então podemos usar gzip para descomprimir este arquivo:

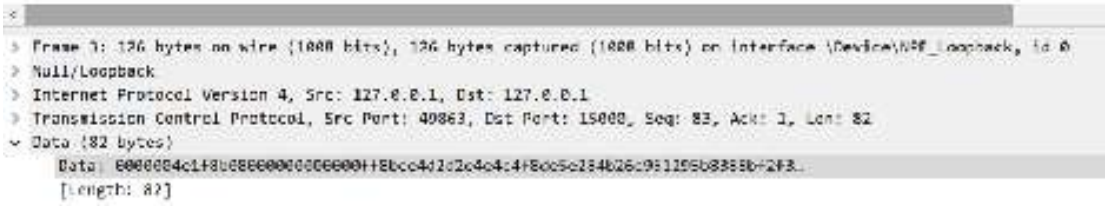
```
Arquivo gzip-d.gz
```

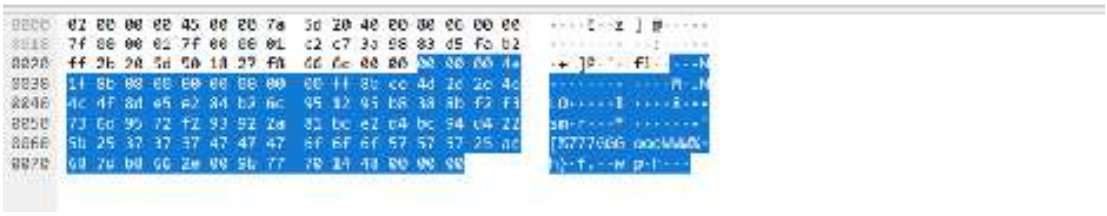
Isso gera um arquivo de texto chamado file. Ao visualizar este arquivo, podemos ver dados estruturados que representam nossas mensagens de bate-papo:

6.3.8 Estrutura de agrupamento

Agora sabemos que a maioria dos pacotes de dados contém dados comprimidos, mas ainda há alguns pacotes que ainda não revertemos. Olhando para o pacote novamente , podemos ver que os dados 0x0000004E aparecem antes da parte comprimida:



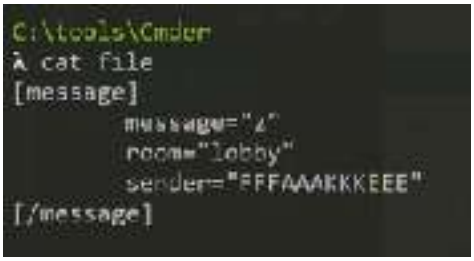




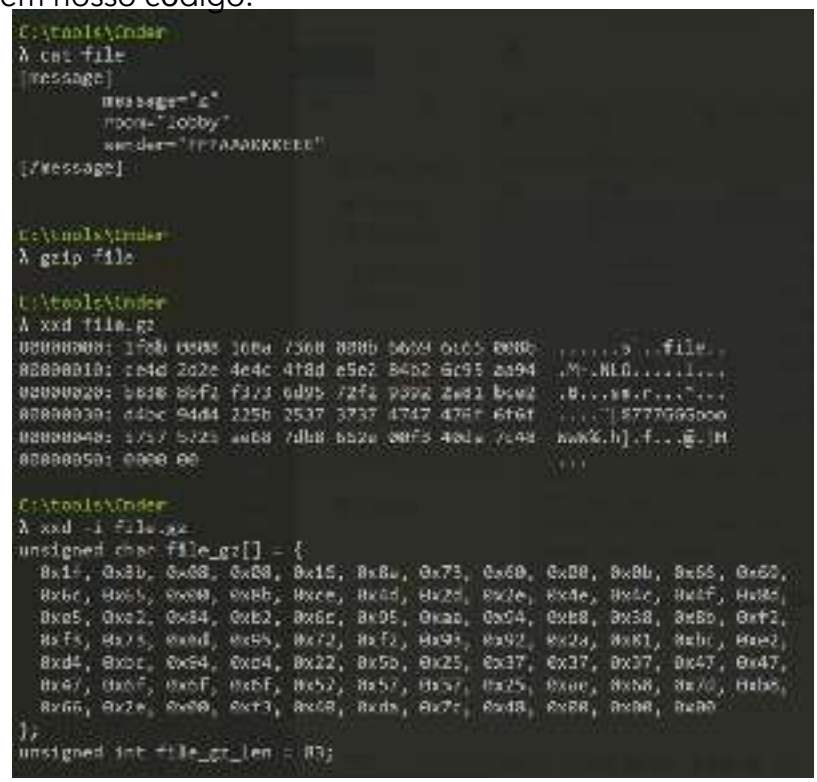
Se convertermos esse 0x4E em decimal, obtemos um valor 78. Verificando o comprimento da seção de dados do pacote, vemos que é 82. A partir disso, podemos inferir que os primeiros 4 bytes de um pacote de dados são responsáveis por salvar o tamanho dos dados comprimidos. Com isso, temos todas as informações necessárias para criar nossos próprios pacotes de dados.

6.3.9 Criar pacotes de dados

Agora que revertemos um pacote, podemos usar as etapas opostas para criar nossos próprios pacotes . Neste caso, vamos criar uma mensagem de bate-papo que representa z a partir da mensagem de bate-papo que representa a. Obtenha o arquivo gerado nas etapas acima e altere a mensagem para z:



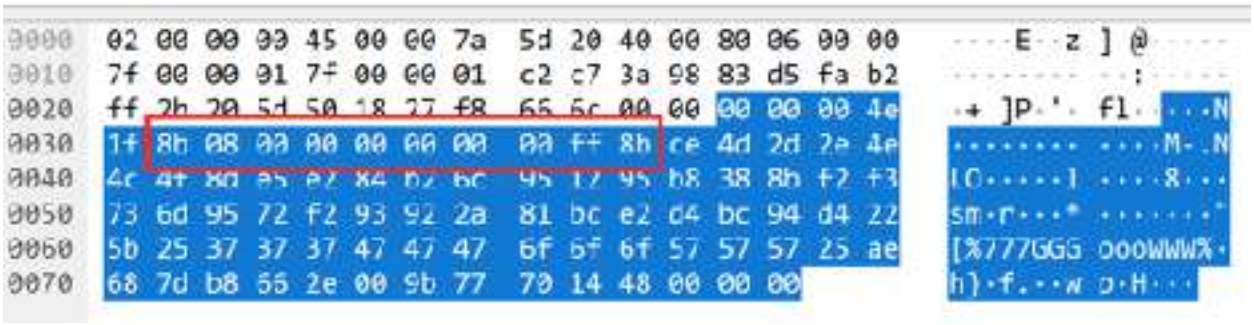
Em seguida, usaremos o arquivo gzip. Podemos então imprimir sua representação de bytes usando xxd. Ao usar a bandeira -i, o xxd exibirá esses dados no formato que podemos usar em nosso código:



Podemos colocar esses dados no código como fizemos antes:

```
const unsigned char packet_bytes[]={
0x1f, 0x8b, 0x08, 0x08, 0x16, 0x8a, 0x73, 0x60, 0x00, 0x0b, 0x66, 0x69, 0x6c, 0x6c, 0x65, 0x00, 0
x8b, 0xce, 0x4d, 0x000002A, 0x0004C, 0x4f, xxxx0bc, 0x0695, 37, 3737, 0x0x0x0x0x0x37, X0x0x0x0
x0x0x0x0x0x37, 0x37, X0x0x37, 0x37, X0x0x0x0x0x37, X0x0x0x0x0x37, X0x37, X0x45, X0x0x0x0x47,
X0x4B, X0x45, X0x45, X0x45, X0x47, X0x4B, X0x0x45, X0x4B, X0x45, X0x45, X0x0x45, X0x45, XX0x45,
X0x45, X0x45, X0x45, x0x0x45, X0x0x45, X0x45, X0x45, x0x45, X0x4B, X0x45, X0x45, X0x0x4B, X0x45
, X0x0x45, X0x45, X0x45, x4B, X0x45, X0x4B, X0x4B, x45, x45, X0x45, x45, X0x45, X0x45, X0x45, X0x
45, x4B, X0x45, X0x45, x45, x4B, x45, X0x45, x4B, x45, x45, x45, x45, X0x45, x45, x45, x45, X0x4B,

};
```

Se você verificar as mensagens de bate-papo que recebemos até agora e as mensagens recém-geradas, poderá notar uma discrepância:



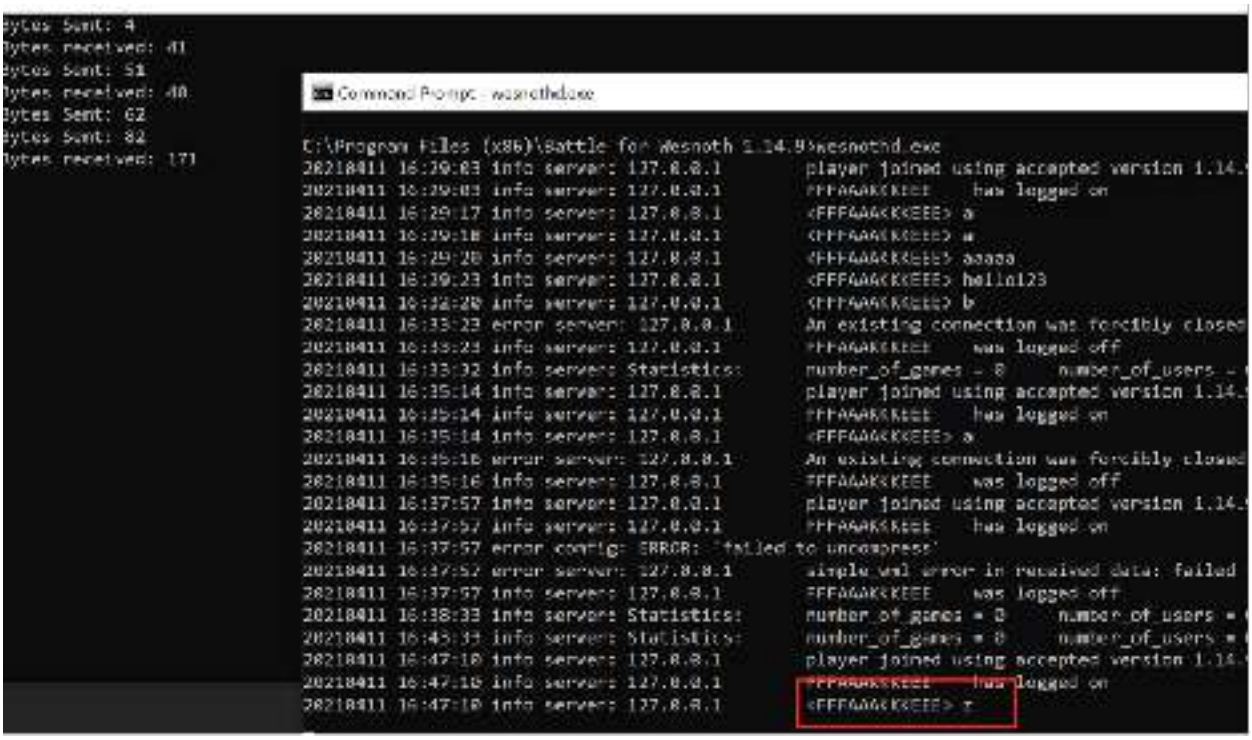
Todas as outras mensagens de bate-papo usadas no jogo têm 0 x 00... ff entre 0x08 e 0x8b. Em contraste, nossas mensagens parecem ser dados aleatórios. Para resolver esse problema , podemos simplesmente substituir esses bytes pelos valores que conhecemos no jogo:

```
const unsigned char packet_bytes[]={
0x1f, 0x8b, 0x08, 0x00, 0x0000, 0x00, 0x00, 0x00, 0xff, 0x8b, 0xce, 0x4d, 0x2d, 0x2d, 0x4d, 0x
4e, 0x000008C, 0x0x00008C, 0x0x08C, 0x047, 37, X0AF, X0695, 37, X037, X0379, XF, X02F, X02F,
X0x04B, X0675, X0x0379, X037, XF, XF, XF, X02F, X0675, X0675, X02F, X037, XF, X04B, X0375.B, X
0675, X0375.B, X04B, X0475.B, X0675, XF, XF, XB, XF, X04B, X045, X0675, X0675, X045, XB, X0475
B, X04B, X04B, X0675, XB, X04B, X04B, X045, X045, X045, XB, XB, X045, XB, X045, XB, X045, X045,
XB, X045, X045, XB, XB, XB, X045, XB, XF, XF, X045, X045, X045, X045, X045, X045, X045, XF, X
045, X045, XB, X045, XB, X045, XF, XF, XF, X045, X045, XB, X045, XF, X045, X045, XF, X045, X045

};
```

Isso também tem o efeito de encurtar os dados. Por fim, precisamos adicionar comprimento à frente do pacote. Como é uma única letra, sabemos que será 0x4e:

```
const unsigned char packet_bytes[]={
0x00, 0x00, 0x00, 0x4e, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
, 0x8b, 0xce, 0x4d, 0x2e, 0x4c, 0x4f, 0x4f, 0X8F, 0X047A, X08B, X09B, 2a




};
```

Com essas mudanças, podemos construir e executar código. O programa gerado se conectará ao servidor e enviará uma mensagem de bate-papo z, provando que agora conhecemos a estrutura para criar nossos próprios pacotes de dados:



6.4 Criação Cliente externo

6.4.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

6.4.2 Identificação

No capítulo anterior, revertemos a estrutura do pacote de bate-papo Wesnoth e determinamos as etapas necessárias para revertê-lo e criar um pacote de bate-papo para o jogo. Neste capítulo, expandiremos esse trabalho para criar um chatbot, um robô que espera e responde a certos comandos.

6.4.3 Compreensão

No capítulo anterior, determinamos o processo de reversão do pacote é o seguinte:

1. Recuperar os dados do pacote de dados.
2. Divida os dados em duas partes: tamanho e conteúdo comprimido.
3. Descomprime o segundo parágrafo.

Usamos essa técnica em pacotes de bate-papo e recuperamos os seguintes dados estruturados:

```
[Mensagem]
Notícias = "A"
Sala = "Salão"
Enviador = "FFFAAKKeE"[/
message]
```

Depois que recuperamos esses dados, podemos modificá-los e criar um novo pacote usando um processo semelhante:

1. Comprimir segmentos de dados.
2. Adicione o comprimento da seção antes da seção de dados.
3. Envie pacotes com novos dados.

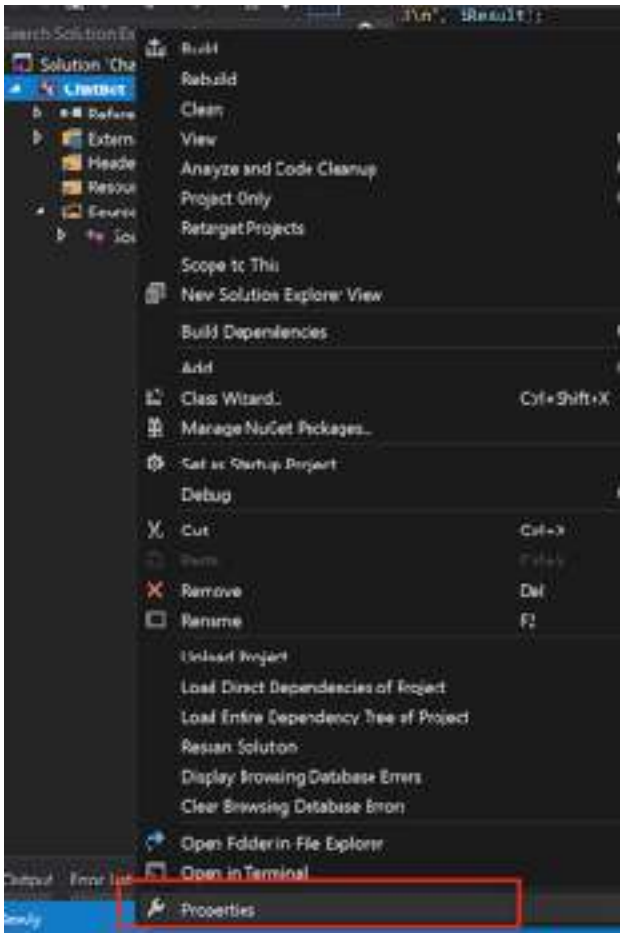
Como podemos escrever os dois processos acima como uma série de etapas específicas, podemos criar um programa para executá-los automaticamente para nós. Durante o processo de recuperação, podemos analisar o conteúdo de cada pacote de dados recuperado e procurar certos caracteres. Se identificarmos esses personagens, podemos agir contra eles.

6.4.4 Instalação ZLib

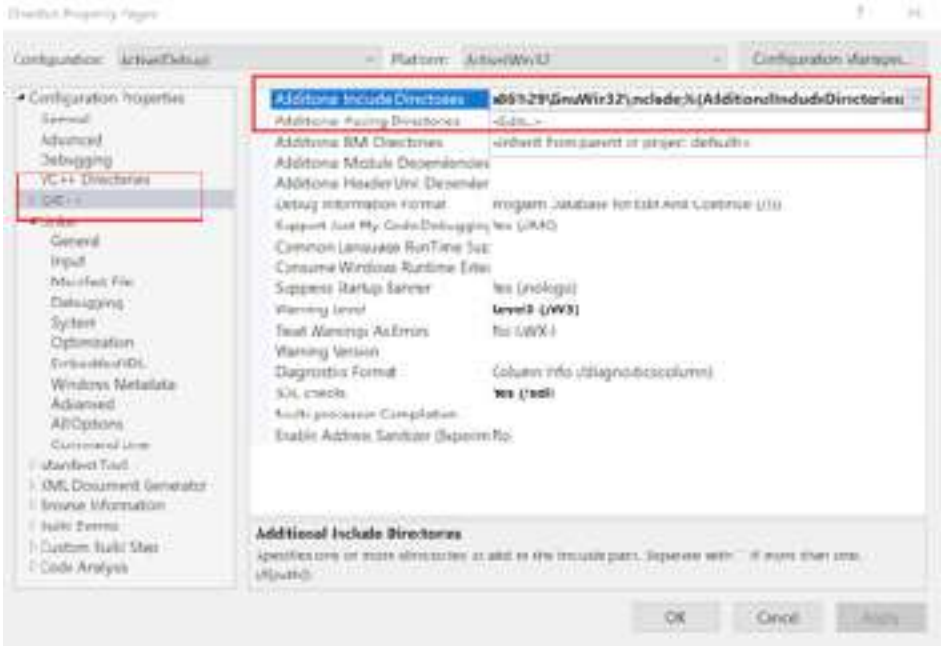
Os dados no pacote de dados são comprimidos usando gzip. Embora possamos escrever nossas próprias funções para gerenciar dados gzip, já existem bibliotecas externas que oferecem a função de comprimir e descomprimir dados gzip em C++. Uma biblioteca externa geralmente consiste em duas partes: um arquivo de cabeçalho contido no código e um arquivo de biblioteca contendo o código real. Neste capítulo, usaremos uma biblioteca chamada ZLib. A maioria dessas bibliotecas exige etapas adicionais de instalação para funcionar totalmente, incluindo o ZLib.

Para configurar o zlib, primeiro baixe a instalação completa do pacote na entrada zlib for Windows 9x/NT em seu site. Após a instalação, ele deve criar um diretório em C:\Program Files(x86)\GnuWin32.

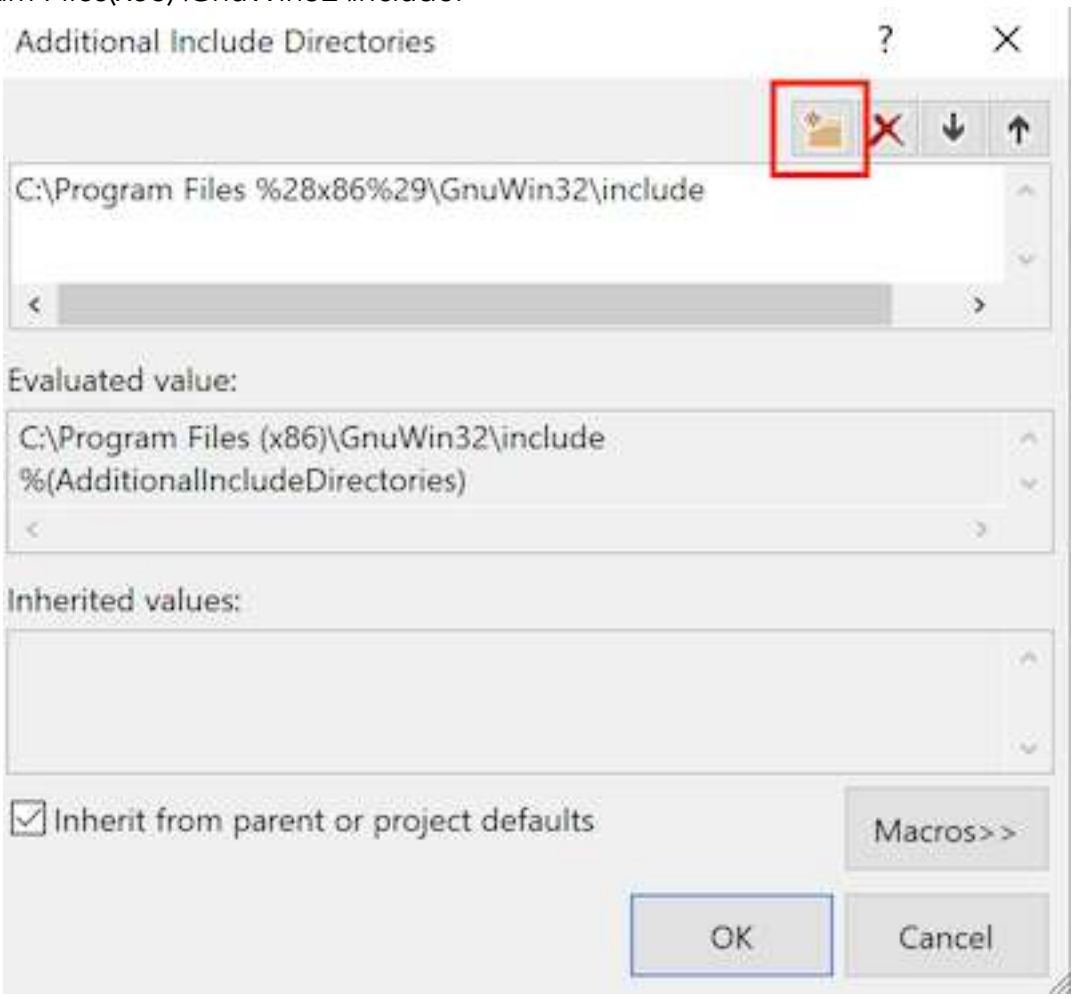
Esta instalação coloca vários arquivos de cabeçalho e arquivos de biblioteca neste diretório. Para usá-los, precisamos incluí-los no projeto Visual Studio. Abra o Visual Studio e crie um projeto. Após a criação, clique com o botão direito no arquivo do projeto e selecione propriedades:



Primeiro, garantiremos que nosso projeto possa encontrar arquivos ZLib include. Na caixa de diálogo de propriedades, selecione C/C++ -> Catálogo de inclusão adicional -> Editar:

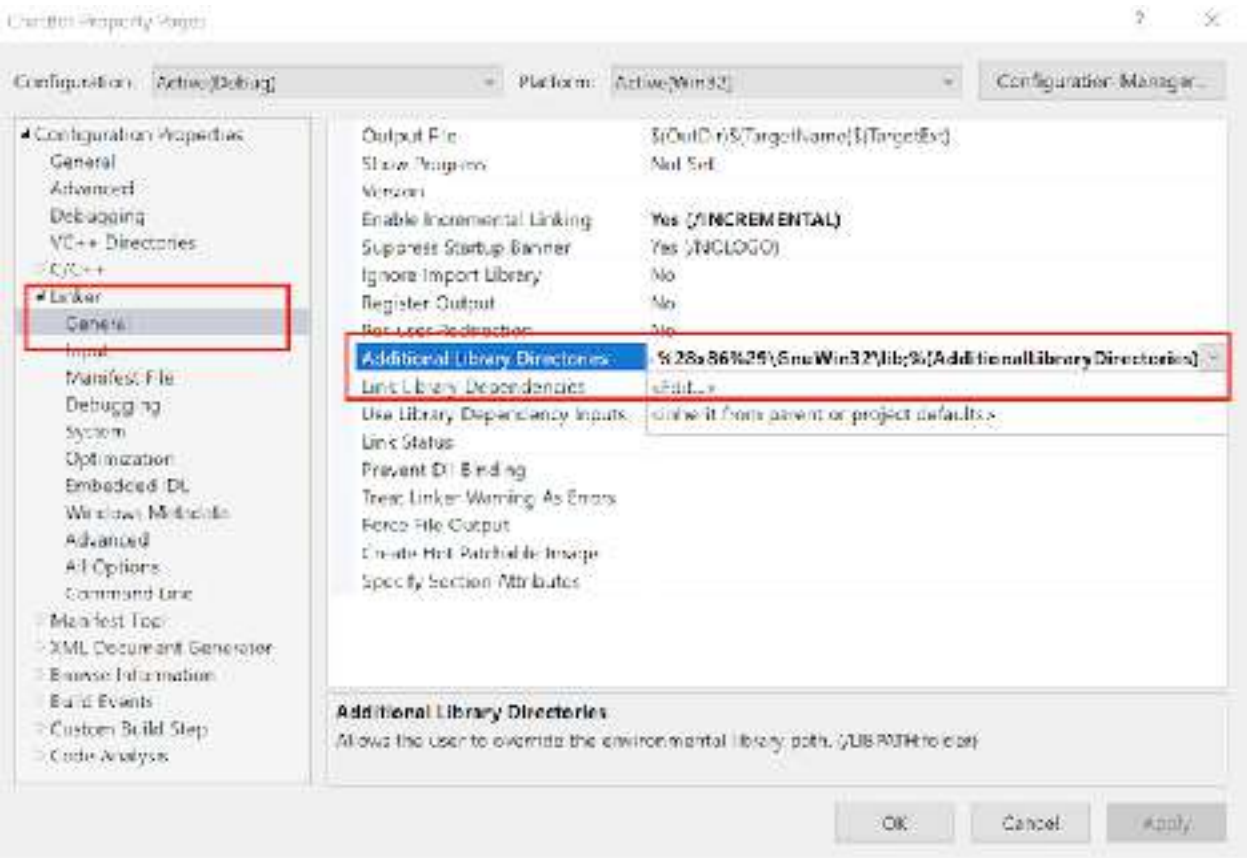


Na caixa de diálogo que aparece, selecione o novo ícone de pasta e navegue para C:\Program Files(x86)\GnuWin32\include:

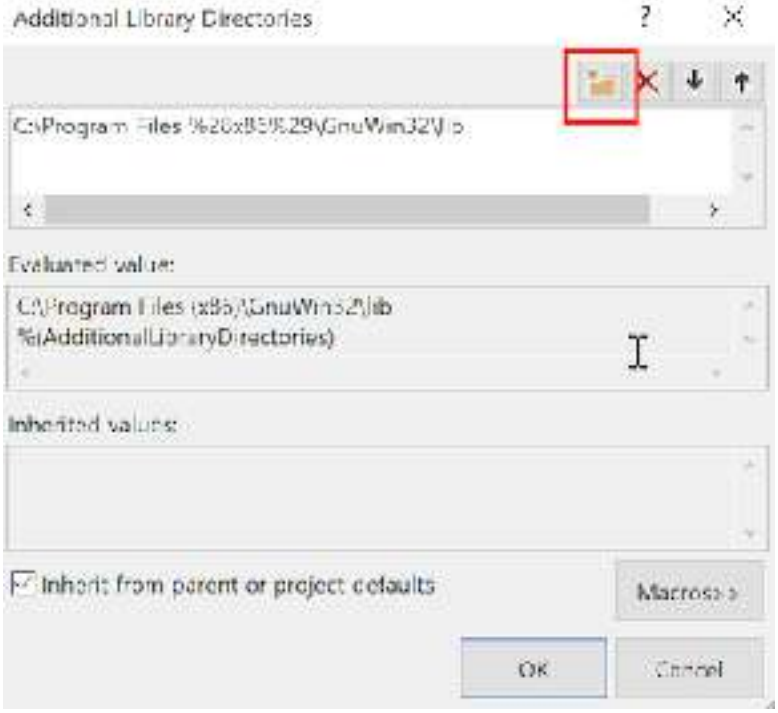


Agora podemos incluir zlib.h no código, e o processo de construção poderá encontrar o arquivo . Entretanto, como vimos nos capítulos anteriores, os arquivos de cabeçalho geralmente contêm apenas definições de função e não a maior parte do código. No ZLib, esse código é armazenado em arquivos de biblioteca. Incluiremos esses arquivos de biblioteca de maneira semelhante.

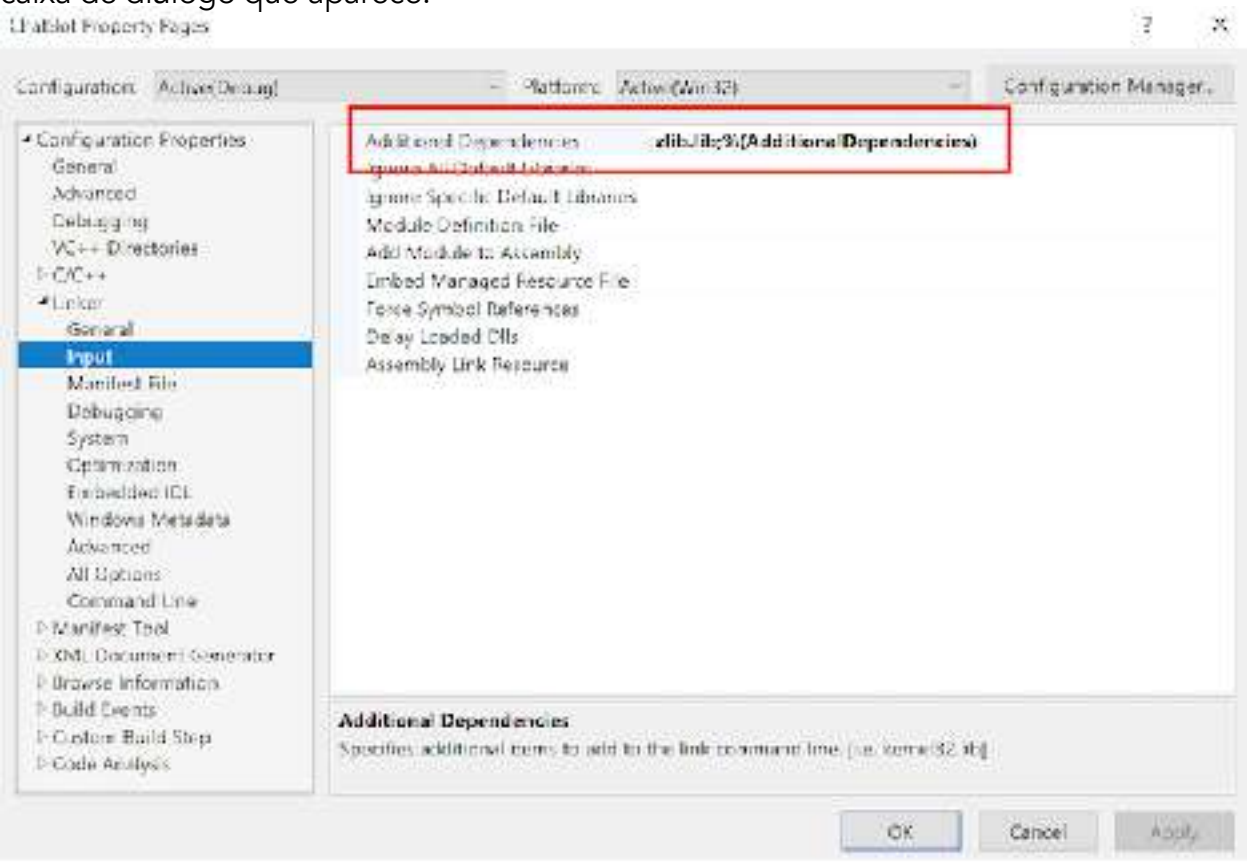
Selecione Linker ->Generial, depois selecione Additive Directories ->Edit:



Assim como acabamos de fazer, selecione um novo ícone de pasta. Desta vez, fornecemos o caminho para a pasta de bibliotecas em C:\Program Files(x86)\GnuWin32\lib:



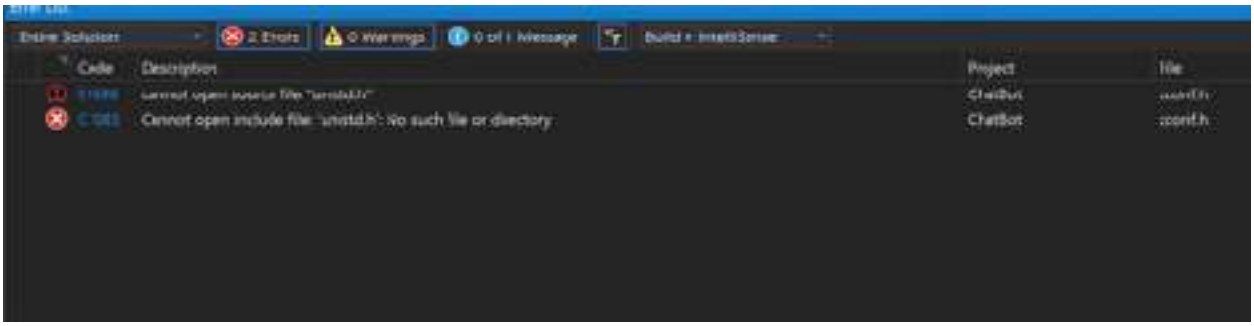
Agora que carregamos a pasta da biblioteca, precisamos incluir o arquivo real. lib. Na entrada do linker ->, selecione uma dependência adicional e adicione zlib.lib na caixa de diálogo que aparece:



Depois de concluir todas essas configurações, agora podemos incluir ZLib em nosso código como um cabeçalho regular:

```
#include<zlib.h>
```

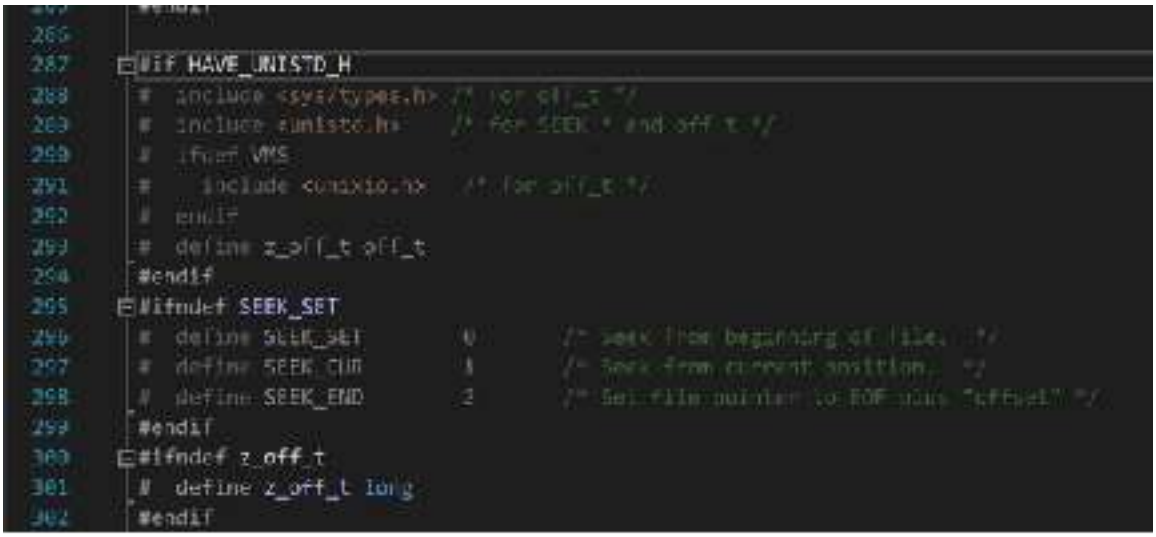
Entretanto, se tentarmos construir esse código, encontraremos os seguintes erros de construção:



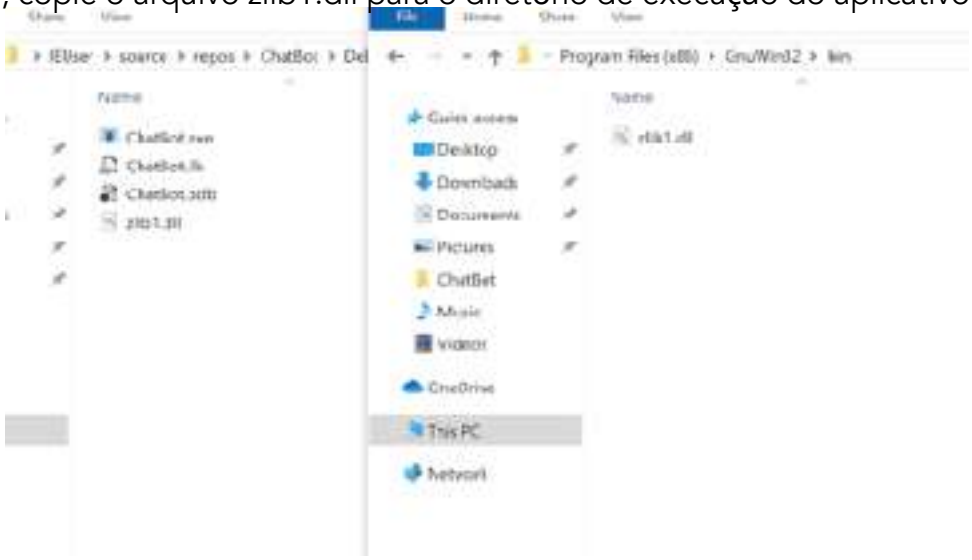
Se você clicar duas vezes neste erro, seremos direcionados para os seguintes blocos de código:

```
#if 1/*Have_Unistd_H--Esta viagem é atualizada por ./configure*/#include<sys/
types.h>/*for offe_t*/
#include<unistd.h>/*for SEEK_* e off_t*/#ifdef VM
```

unistd.h é um arquivo específico do Unix. Para corrigir esse erro, podemos alterar #if 1 para #if HAVE_UNISTD_H, conforme mostrado abaixo:



Com essas mudanças, nosso programa agora será construído. No entanto, se você executar este programa, você encontrará um erro de perda de DLL. Para resolver esse problema, copie o arquivo zlib1.dll para o diretório de execução do aplicativo:



Agora podemos criar e executar programas que contêm funções ZLib.

6.4.5 Enviar dados

Como podemos comprimir dados em nosso código, podemos criar uma função para enviar um pacote de dados estruturado Wesnoth contendo quaisquer dados que desejamos. Para criar pacotes de dados, essa função requer nossos dados e o comprimento dos dados. Para enviar pacotes de dados, precisamos de um socket. Com esses requisitos, podemos criar nossa definição de função:

```
void send_data (condens*data, tamanho _t len, SOCKET s){
```

Dependendo da documentação, a maneira mais fácil de gerar dados gzip usando ZLib é criar um arquivo contendo dados comprimidos. Podemos fazer isso usando as funções gzopen, gzwrite e gzclose. Essas funções são semelhantes às funções de arquivo convencionais FOEN, fwrite e fclose. Em nosso exemplo, criaremos um arquivo compactado, packet.gz, e escreveremos quaisquer dados transferidos para o arquivo em:

```
gzFile templ_data=gzopenen ("packet.gz", "wb
"); gzwrite (temp_data, data, len);
gzclose (temp_data);
```

Podemos testar a implementação atual das seguintes maneiras:

```
const sem sinal versão []="[test]Olá[/test]"; send_data
(versão, versão);
```

Executar este código gerará um arquivo packet.gz no mesmo diretório do programa de execução. Se você usar gzip para descomprimir o arquivo packet.gz, descobrirá que ele contém [test] hello[/test], o que significa que nosso código está funcionando até agora.

Nosso pacote de dados precisa conter a representação de bytes deste arquivo. Para recuperá-lo, podemos ler o arquivo como um arquivo binário:

```
#define DEFUL_BUFLEN 512
...
file*temp_filo=NULL;
fopen_s(&temp_filo, "packet.gz", "rb");
```

```
if(temp_filo){
size_t compress_len=0;
Buffer de caracteres sem símbolos [DEFAULT_BUFLEN]={0}; Comples_
len=read (buffer, 1, tamp_filer); fclose (temp_filme);
```

Se você executar esse código e definir um ponto de interrupção após o fclose, verá que o buffer agora contém uma representação de byte do arquivo compactado. Podemos usá-lo para construir nossos pacotes de dados. Sabemos que os primeiros 4 bytes (um DWORD) de um pacote de dados representam o tamanho do pacote de dados. Como todos os pacotes de bate-papo são muito pequenos, precisamos apenas escrever o tamanho no último byte deste DWORD. Como o buffer começa com 0 em C, referimos essa posição com +3. Em seguida, gravaremos um buffer que contém dados compactados:

```
unsigned char buff_packet[default_buffen]={0}; memcpy (buff_
packet+3, &compress_len, tamanho (compress_len)); memcpy (buff_
packet+4, buffer, compress_len);
```

Em seguida, usaremos o código que vimos anteriormente para enviar pacotes contendo os seguintes dados:

```
int iResult=send(s, (const char*)buff_packet,compress_len+4,0); printf
("Bytes enviados: %ld\n", iResult);
```

Para verificar se essa abordagem é eficaz, construiremos com base no código escrito no capítulo anterior. Neste código, enviamos três pacotes de dados para se conectar ao servidor: um pacote de negociação inicial contendo 0, um pacote de dados contendo a versão do cliente e um pacote de dados contendo o nome do usuário.

Podemos decodificar esses pacotes usando as mesmas técnicas que decodificamos mensagens de bate-papo. Por exemplo, o pacote que contém nossa versão é decodificado como este:

```
[Versão]
version="1.14.9"[/
version]
```

Podemos usar novas funções em vez de enviar bytes de pacotes de dados como antes:

```
const unsigned char version[]="[version]\nversion=\"1.14.9\"\n[/version]";
send_data (versão, versão);
```

Se você construir e executar esse código, verá que nossos robôs se conectarão da mesma maneira para verificar se nossas funções são válidas. Podemos enviar nomes de usuário personalizados com base nisso:

```
const unsigned char Nome[]="[login]\nusername="ChatBot\"\n[/login]"; send
_data (nome, sizeof (nome), conect);
```

Por fim, podemos usar o mesmo método para enviar a mensagem inicial de bate-papo durante a conexão:

```
const Unsigneeded char first_message[]="[message]\Nessage=\"ChatBot
Connect\"Nessage=\"Nessage=\"Nessage=\"ChatBot\"Nessage=\"ChatBot\"
Nessage"; send_data (first_message, sizeof (first_message), Conect);
```

Esta é a mesma estrutura da mensagem de bate-papo que observamos no capítulo anterior.

6.4.6 Recuperação de dados

Agora podemos enviar mensagens de bate-papo. Para fazer nossos chatbots funcionarem, também precisamos recuperar mensagens do servidor e analisar alguns de seus textos. Neste capítulo, pediremos aos nossos robôs que respondam a qualquer mensagem contendo\wave usando uma mensagem de bate-papo e os cumprimentem.

Há um loop na parte inferior do código da Microsoft que verifica constantemente novos pacotes de dados. Podemos modificá-lo enviando pacotes recuperados para nossas próprias funções e enviando mensagens de bate-papo com base no conteúdo:

```
De {
iRecort = Recort(Conect, (char*)Recort, Recort, 0); Se (iResult>0)

printf ("Bytes recebidos: %d\n", iResult);
Caso contrário, se (iResult===0)
printf("Conexão fechada\n");
else
printf ("recv falhou, erro %d\n", wsagetlasterrol());

Se(parse_data (recvbuf, iResult)){
const unsigned char message[]="[message]\Nessage=\"Olá! \"\nroom=\"
lobby\"\nsender=\\"ChatBot\"\n[/message]"; send_data (message,
message (message), conecticlock);
}
} while(iResult>0);
```

Nossa função parse_data obtém um buffer de dados e seu comprimento e, se for encontrado\wave, retorna verdade.

```
bool parse_data (unsigned char*buff,int buff_len){
```

Como descrevemos acima, a função executa as mesmas etapas da função send_data, mas na ordem oposta. Primeiro, extrairemos dados comprimidos do pacote de dados e os gravaremos no arquivo:

```
Dados de caracteres sem símbolos [DEFAULT_BUFLEN
]={0}; memcpy (dados, buff+4, buff_len-4);

file*temp_filo=NULL;
fopen_s(&temp_filo, "packet_recv.gz", "wb");

if(temp_filo){
fwrite (data, 1, sizeof (data), temp_file);
fclose (temp_filme);
}
```

Depois de salvar os dados comprimidos, podemos usar gzopen e gzread para ler os dados descomprimidos em uma variável. Em seguida, usaremos fwrite para escrever esta variável no terminal:

```
gzFile temp_data_in=gzopenen ("packet_recv.gz", "rb");
unsigned char descompresse_Data[defaut_buflen]={0};
gzread (temp_data_in, decompresse_data, default_bufflen);
fwrite (decompresse_data, 1, default_bufflen, estoque);
gzclose(temp_data_in);
```

Por fim, usaremos o StrTR para verificar se os dados contêm texto\wave. Se o segundo parâmetro estiver incluído no primeiro parâmetro, a função retorna um valor positivo. Caso contrário, retorna a 0. Portanto, podemos devolver o valor da pesquisa e usá-lo para indicar ao código de chamada que o texto foi encontrado:

```
Retorne a strstr((const char*)decompresse_data, (const char*) "\\\wave");
```

O código-fonte completo deste capítulo pode ser comparado no Apêndice A.

6.5 Agente TCP Transporte

6.5.1 Objetivo

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

6.5.2 Visão geral

No capítulo anterior, criamos um cliente externo que se conectará ao servidor Wesnoth e monitorará e responderá a mensagens de bate-papo específicas. Uma das principais desvantagens dessa abordagem é que temos que reverter todo o processo de autenticação para que nosso cliente possa se conectar ao servidor. Nosso objetivo neste capítulo é criar um agente. Isso nos permitirá usar um cliente de jogo regular e interceptar e modificar apenas o tráfego com o qual nos importamos do cliente.

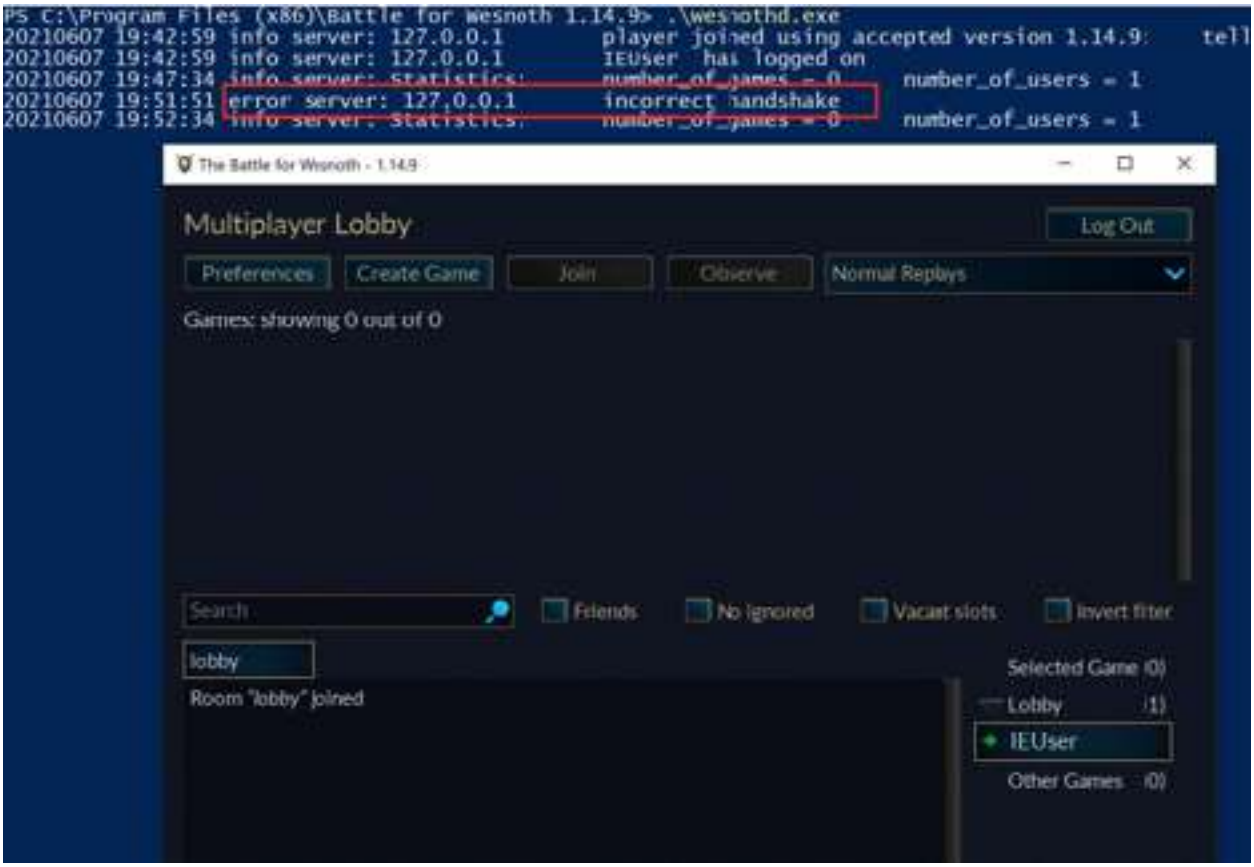
6.5.3 Motivos da Agência

A melhor maneira de entender o propósito de criar proxy é observar o tráfego de rede ao se conectar a um servidor com dois clientes no mesmo host. Em sua máquina de laboratório, inicie um servidor Wesnoth e conecte-se a ele usando uma cópia legal do jogo . Em seguida, modifique o cliente que escrevemos no capítulo anterior e exclua todas as autenticações. Em vez disso, deixe-o simplesmente enviar uma mensagem de bate-papo:

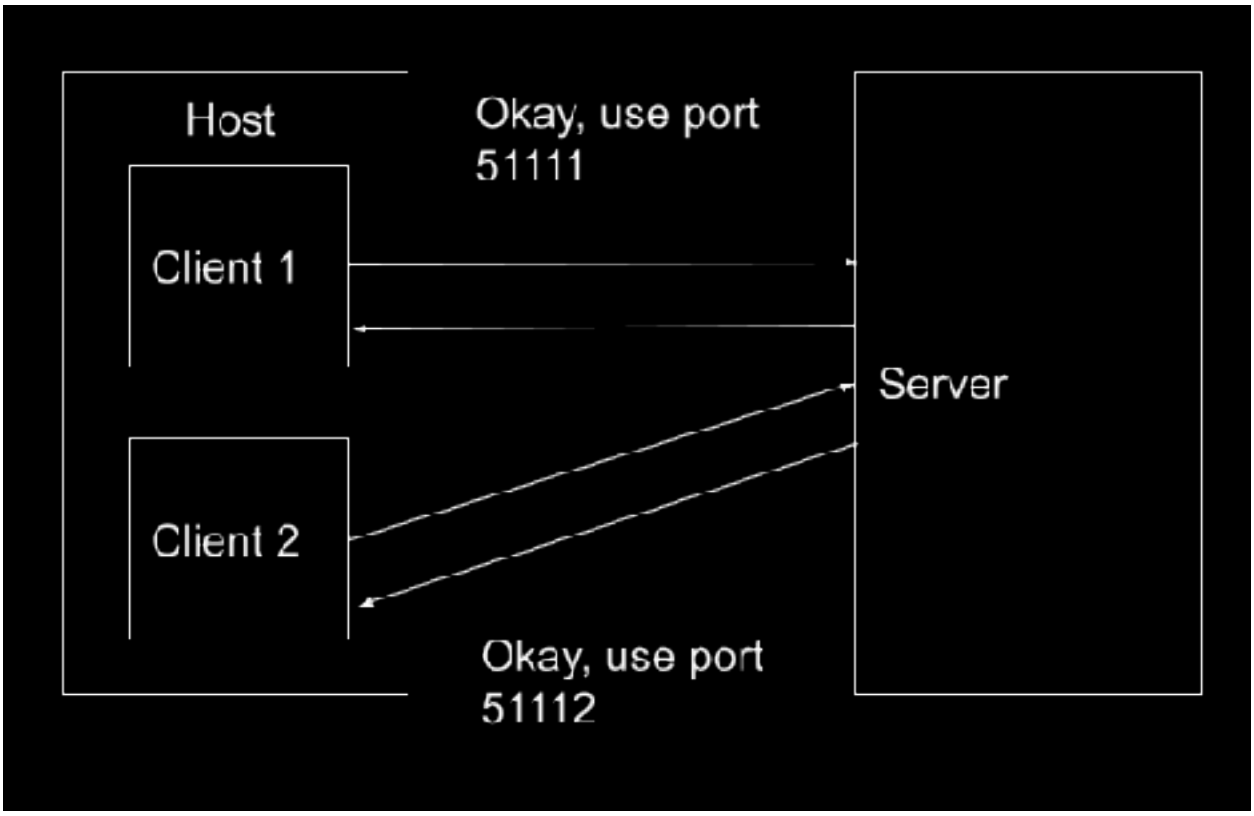
```
...
freeaddrinfo (resultados);

Se(conective_Socket==INVALID_SOCKET){
printf ("Não consegue se conectar ao servidor!\n
"); wsaacleanup ();
Retornar 1;}
```

```
const Unsigneeded char first_message[]="[message]\Nessage=\"ChatBot
Connect\"Nessage=\"Nessage=\"Nessage=\"ChatBot\"Nessage=\"ChatBot\"
Nessage"; send_data (first_message, sizeof (first_message), Conect);
…
```

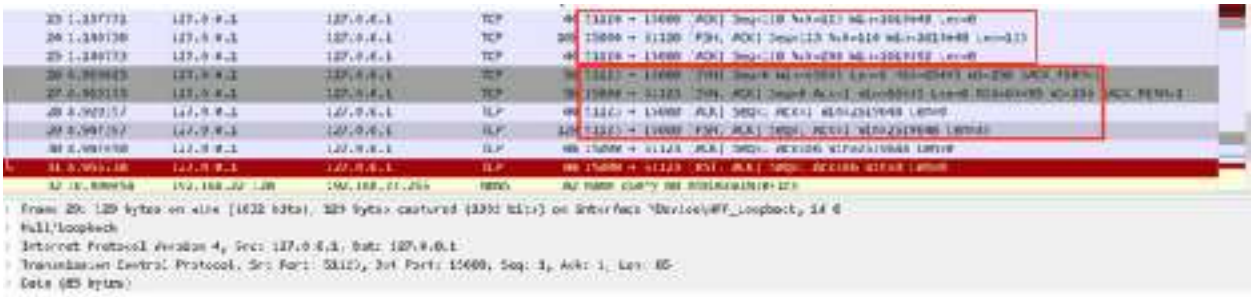
Por fim, ligue o Wireshark e comece a monitorar o tráfego no adaptador local, como discutimos em capítulos anteriores. Ao iniciar um cliente externo modificado, você deve ver o seguinte erro no servidor:



Estamos familiarizados com esse erro em nossa análise inicial, então sabemos que ele acontece quando não oferecemos um aperto de mão eficaz. Mesmo que nossos jogos e clientes externos sejam executados na mesma máquina, eles têm sockets diferentes e são considerados conexões completamente independentes pelo servidor:



Se verificarmos o tráfego Wireshark de clientes de jogos e clientes externos, podemos observar esse comportamento. Quando o jogo está ocupado enviando tráfego na porta 51120, podemos ver o aperto de mão TCP do nosso cliente externo na porta 51123:

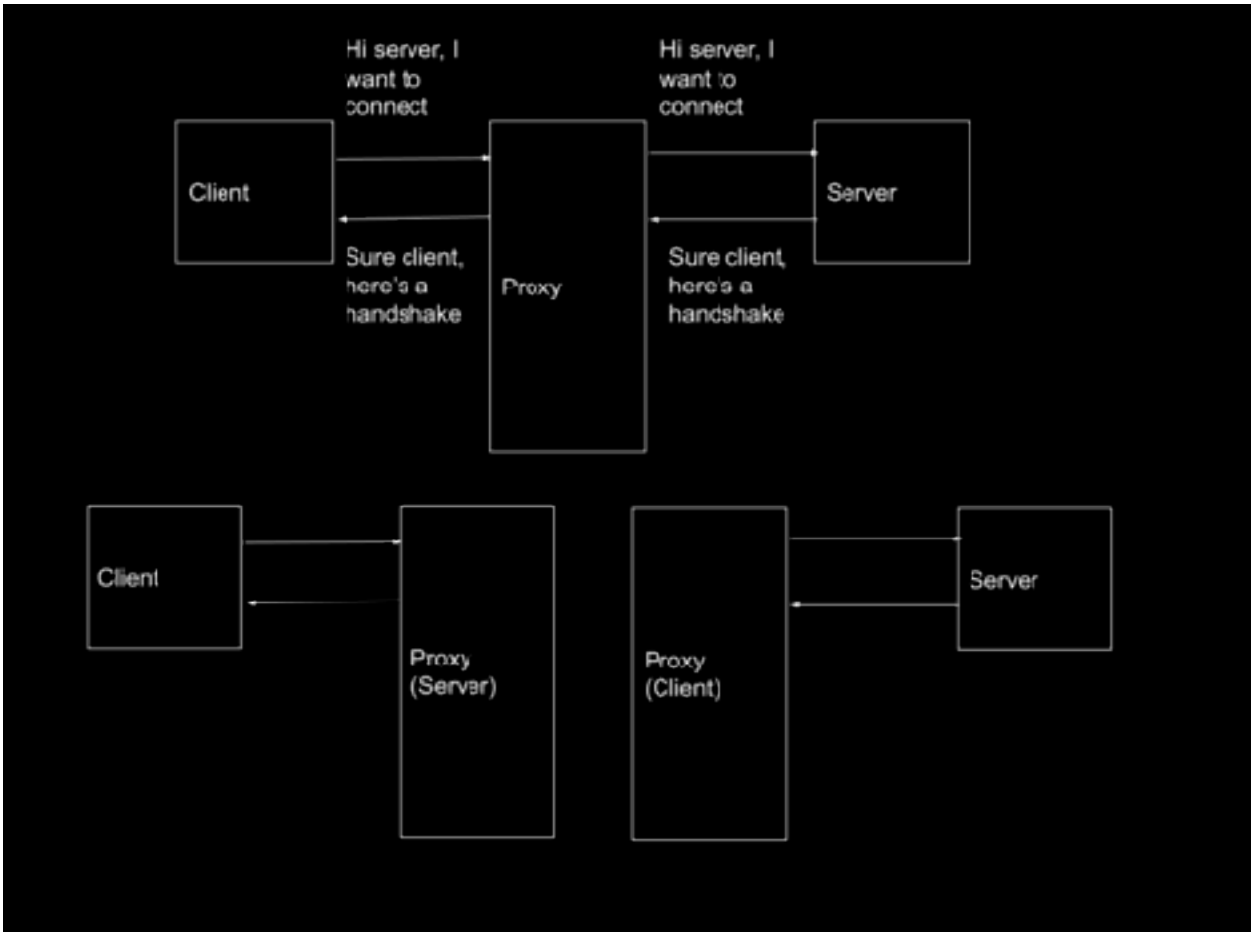


Se quisermos interceptar e modificar o tráfego do cliente do jogo, precisamos usar uma abordagem diferente.

6.5.4 Tráfego do agente

Ao usar o TCP, sabemos que uma conexão é estabelecida entre o cliente e o servidor após a conclusão do aperto de mão. A maneira mais fácil de interceptar e injetar nosso próprio tráfego nesta conexão é nos tornar o intermediário (MitM) dessa conexão.

Neste local, podemos modificar a solicitação do cliente antes de enviá-la ao servidor, ou podemos modificá-la antes que a resposta do servidor seja enviada ao cliente. Isso é geralmente chamado de proxy, ou um proxy que simplesmente transmite tráfego da fonte para o destino. A visualização deste modelo é a seguinte:



A chave para esse modelo é que nosso proxy atua como servidor para cliente e cliente para servidor. Isso significa que o servidor completou o aperto de mão com nossos agentes, permitindo-nos injetar qualquer tráfego que quisermos. Como estamos encaminhando tráfego legal do cliente, não precisamos de tráfego reverso (como mecanismos de autenticação) porque o cliente processará esse tráfego para nós.

O agente consiste em três partes principais de código:

1. Sockets para monitorar o tráfego do cliente
2. Socket para enviar tráfego ao servidor
3. A lógica do tráfego de retransmissão do cliente ao servidor e do servidor ao cliente

Para simplificar este capítulo, criaremos um proxy para responder ao evento\wave, o mesmo cliente externo que escrevemos no capítulo anterior. Para essas operações, o cliente Wesnoth deve enviar dados ao servidor para que ele possa responder. Ao usar um proxy para realizar outras operações, lógica adicional precisa ser adicionada para transferir o tráfego do servidor para o cliente.

O código completo para este capítulo está disponível no Apêndice A.

6.5.5 Monitorar o tráfego do cliente

Para monitorar o tráfego do cliente, criaremos um socket de monitoramento. Quando recebemos uma conexão, estabelecemos uma conexão com o cliente. Para isso, podemos construir um exemplo do Microsoft server:

```
#define DEFULT_Porto "27015"

WSADA WSADA;
int iResult

Lista de sockets sockets = invalid_sockets;
Socket cliente Socket = inválido_Socket;

struct addrinfo*result=NULL,
Dicas;

iResult = WSA Startup (2,2), &WSA Data);

Zeromory (hints, hints);
hints.ai_family=af_int;
hints.ai_socktype=SOCK_STREAM;
hints.ai_protocol=IPPROTO_TCP;
hints.ai_flags=AI_PASSIVE;

iResult=getaddrinfo (NULL, DEFAULT_PORT, &hints, &RELT); ListenSocket=
ListenSocket (ListenSocket = ListenSocket = ListenSocket (ListenSocket);
```

```
Iresolt=bind(ListenSocket, resolt->ai_addr, (int) resolt->ai_addren);
freeaddrinfo (resultados);

Iresolt=listen (ListenSocket, SOMAXCONN);
ClientSocket = acept (Listentsocket, NULL, NULL);
Closeocket (ListenSocket);
```

Isso criará um socket na porta 27015 que aceita uma única conexão.

6.5.6 Enviar tráfego para o servidor

Para enviar tráfego ao servidor, podemos criar o código discutido no capítulo anterior:

```
SOCKET ServerSocket=INVALID_SOCKET;

Zeromory (hints, hints);
hints.ai_family=af_int;
hints.ai_socktype=SOCK_STREAM;
hints.ai_protocol=IPPROTO_TCP;

iResult=getaddrinfo ("127,0,0,1", "15000", &hints, &result);
Verserocket=socket (result->ai_family, result->ai_socktype, result-
>ai_protocol);
iResult = Conect (Serversoft, result ->ai_addr, (int)result ->Ai_Addler);
freeaddrinfo (resultados);
```

Como discutimos acima, nosso proxy encaminhará pacotes de dados do cliente para o servidor e encaminhará pacotes de dados do servidor para o cliente. Entretanto, nem todos os pacotes de dados do cliente exigem uma resposta do servidor. Por exemplo, no Wesnoth, enviar mensagens de bate-papo não requer resposta. Para garantir que nosso proxy não fique travado enquanto aguarda a resposta do servidor, precisamos definir um limite de tempo no socket do servidor. Este limite de tempo fará com que qualquer chamada de recv falhe após o horário definido:

```
Tempo limite do DWORD = 1000;
SETSOCKET (SETSOCKET, SO_RCVTIMEO, (char*)&timeout, SETEOUT (timeout
);
```

6.5.7 Tráfego de retransmissão

Com a criação de dois sockets, agora podemos nos concentrar no tráfego entre clientes de retransmissão e servidores. Como o Wesnoth não emite uma resposta do servidor, a menos que o cliente envie uma solicitação, nosso proxy pode ser simplificado para os seguintes eventos:

1. Aguarde a solicitação do cliente.
2. Envie a solicitação para o servidor.
3. Aguarde a resposta do servidor.
4. Se uma resposta for retornada, envie-a ao cliente. Caso contrário, comece a aguardar a próxima solicitação do cliente.

Após cada evento, nosso programa ficará inativo por um curto período de tempo para garantir que o tráfego entre o cliente e o servidor não esteja dessincronizado. Primeiro , aguardaremos a solicitação do cliente:

```
#define DEFUL_BUFLEN 512

int iSendResult;
Caracteres não símbolos recvbuf[DEFAULT_BUFLEN];
int recvuflen=DEFULT_BUFLEN;

De {
Iresuult = recv(ClientSocket, (char*)Recvbuff, Recvbufflen, 0); Sono (
100);
```

Se recuperarmos uma solicitação, passamos esses dados para o servidor:

```
Se(iResult>0){
printf ("Bytes recebidos: %d\n", iResult);

iSendResult = send (server socket, (char*)recvbuf, isult, 0); Sono (
100);
printf ("Bytes enviados: %d\n", iSendResult);
```

Em seguida, aguardamos a resposta do servidor. Se uma resposta for realmente retornada, nós a encaminhamos para o cliente:

```
iResult=recv(ServerSocket, (char*)Recvbuff, Recvbufflen, 0);
Sono (100);
```

```
Se(iResult!=SOCKET_ERROR){
iSendResult=send (ClientSocket, (char*) recvbuf, isult, 0); Sono (
100);
}
```

Por fim, quando obtivemos o resultado, ou se tivermos um limite de tempo do servidor, continuamos com o loop:

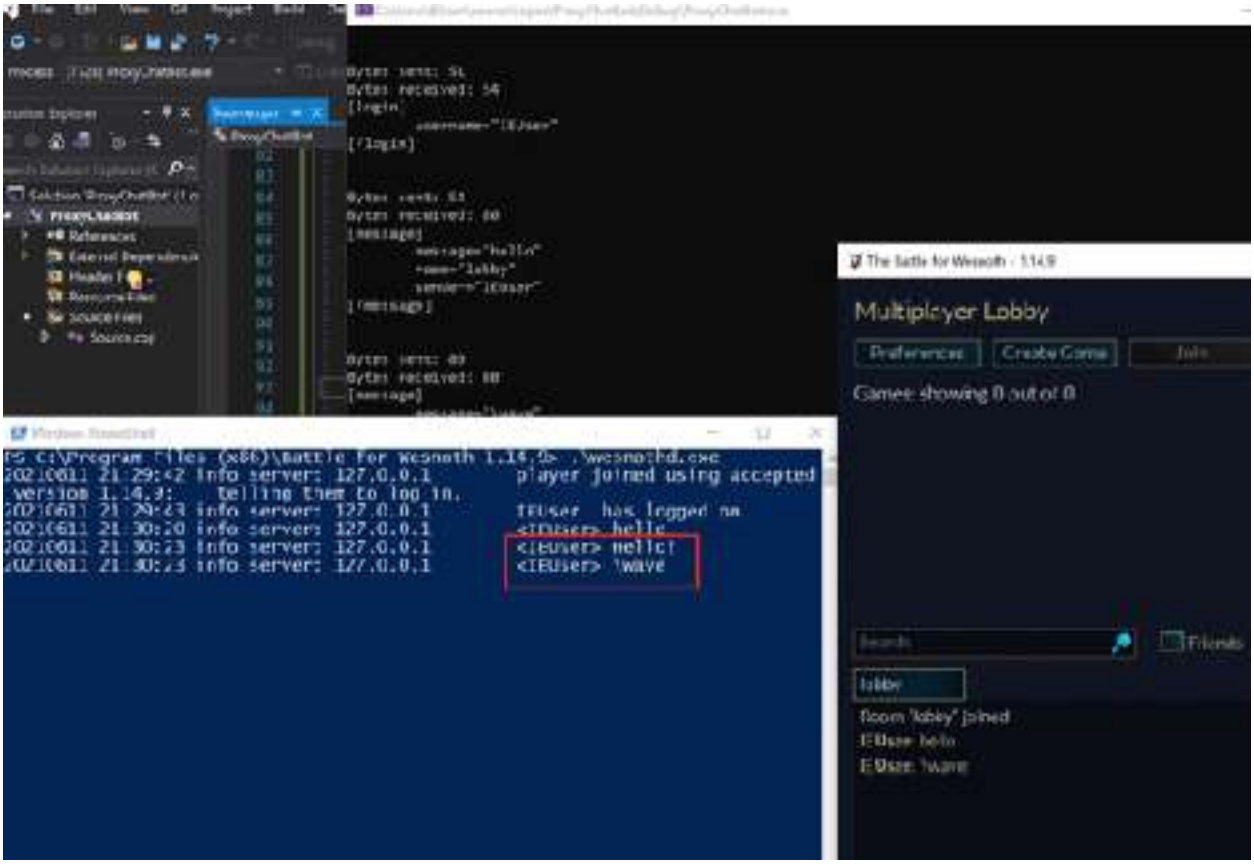
```
}while(iResult>0||WSAGETLETEROR()==WSAETEROUT);
```

Neste ponto, nosso proxy transferirá corretamente o tráfego do cliente para o servidor. Podemos verificar isso por meio de um proxy executando, configurando o servidor para localhost: 27015 e conectando-se a ele no Wesnoth e confirmando que podemos conectar-se ao servidor real executado em localhost: 15000.

Como prova de conceito, agora podemos adicionar lógica para interceptar e modificar o tráfego. Primeiro , podemos importar as funções parse_data e send_data do capítulo anterior. Em seguida, modificaremos nosso loop principal para verificar quaisquer solicitações do cliente para ver se elas contêm mensagens de bate-papo\wave. Se for esse o caso, enviaremos um pacote adicional com você! Informações:

```
Se(iResult>0){
printf ("Bytes recebidos: %d\n", iResult);
Se(parse_data (recvbuf, iResult)){
const unsigned char message[]="[message]\Nessage=\"Olá! \"\nroom=\"
lobby\"\nsender=\\"ChatBot\"\n[/message]"; send_data (message,
message (message), ServerSocket); Sono (100);

}
```

Se você se conectar ao proxy agora e enviar uma mensagem de bate-papo\wave, verá uma mensagem adicional no servidor indicando que injetamos tráfego com sucesso na conexão:


Multiplayer Lobby
Preferences
Create Game
Games showing 0 out of 0
Friends
lobby
Room 'lobby' joined
EBUser: hello
EBUser: !wave

Parte 7
Ferramentas Desenvolvimento

7.1 Injetor DLL

7.1.1 Metas

Ao escrever um injetor de DLL, é útil preparar um DLL que já está funcionando para um objetivo específico. Neste capítulo, usaremos o hacker de parede de memória que criamos no Capítulo 5.2. Embora nossas seringas sejam construídas para o jogo City Horror, seremos capazes de modificá-las facilmente para outros objetivos no futuro.

7.1.2 Visão geral

Nos capítulos anteriores, usamos a função AppInit da Windows para injetar DLL no arquivo executável do jogo. Embora esse método seja adequado para testes, ele tem várias desvantagens:

- AppInit_DLLs precisa ser atualizado para cada novo DLL.
- appinit_dll é injetado em cada processo iniciado.
- A orientação de segurança deve ser desativada.

•appinit_dll será injetado somente no processo de carregamento do user32.dll.
•DLL é carregado no processo no momento definido, além do nosso controle.

Para contornar essas deficiências, escreveremos um injetor que carregará manualmente nosso DLL no arquivo executável do jogo.

7.1.3 Conceito

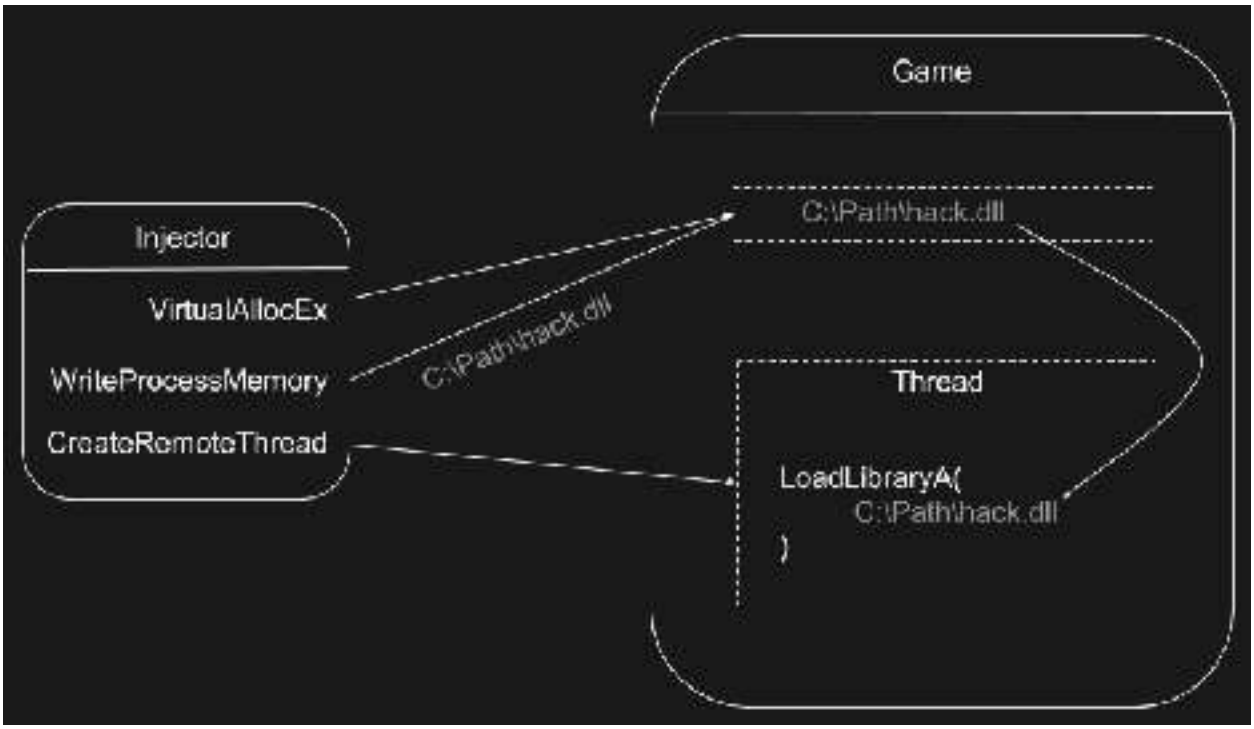
Para carregar bibliotecas estáticas e dinâmicas, os executáveis Windows podem ser usados usando a função API Loadlibrearya. Esta função aceita um parâmetro que é o caminho completo para a biblioteca a ser carregada.

```
HMODULE LoadLibraryA (
LPCSTR LPLIBLEAME);
```

Se chamarmos Laoadlibrearya no código do injetor, o DLL será carregado na memória do injetor . Em vez disso, queremos que nossos injetores forcem o jogo a chamar o loadlibraya. Faça isso

Para isso, criaremos um novo thread no jogo usando a API Creater EMOTETHEAD. O thread então executa o loadlibrearya enquanto o jogo está em execução.

Entretanto, como os threads estão executando na memória do jogo, o Ladlibrearya não conseguirá encontrar o caminho para o DLL especificado no injetor. Para resolver esse problema, precisamos gravar o caminho do DLL na memória do jogo. Para garantir que não danifiquemos nenhuma outra memória, também precisamos usar o VirtuAllocex para alocar memória adicional no jogo. Uma decomposição completa dessa interação é a seguinte:



Como sabemos em capítulos anteriores, precisamos de um cabo de processo para interagir com processos externos. Por exemplo, no Capítulo 3.2, usamos FindWindows e GetWindows ReadProcess para recuperar identificadores de processo. Este método tem muitas desvantagens e não é recomendado, exceto para testes rápidos. Em vez disso, usaremos o Creatotolhelp32Snapshot.

7.1.4 Identificador de Processo

Para usar o WriteProcessMemory, precisamos de uma alavanca para o processo de terror urbano. Não usamos FindWindows como antes, mas usamos

Crie uma ferramenta para ajudar 32 instantâneo. Esta API captura instantâneos de todos os processos atualmente em execução na máquina. Você pode então verificar cada processo neste instantâneo usando o seguinte método

O primeiro processo 32 e o próximo processo 32. Aqui, a Microsoft oferece um bom exemplo de como isso é feito.

Embora os exemplos do Microsoft iterem todos os processos e despejam os módulos que eles carregam, estamos interessados apenas em encontrar processos individuais e recuperar seus identificadores de processo. Portanto, podemos simplificar seu código de exemplo:

```
#include<windows.h>
#include<tlhelp32.h>

int main(int argc, char**argv){
	Instantâneo do cabo = 0;
	processEntry32 pe 32={0};

	pe32. dwSize = dwSize (PROCESSENTRY32);
	snapshot=Creatotolhelp32Snapshot (Th32cs_Snapshot, 0);
	process 32 first (instantâneo, &P32);

	De {

	}while (Process32Next (instantâneo, &pe32
)); Retornar a 0;

}
```

Cada entrada de processo contém dois campos com os quais nos importamos: SzEFEILE e ID do processo TH32. O primeiro contém o nome do processo, como svchost.exe ou notepad.exe. Este último contém identificadores de processo que podemos passar para o processo OpenProcess.

O nome do processo de terror urbano é Quake3-UrT.exe. Isso pode ser identificado visualizando a lista de processos no gerenciador de tarefas enquanto o City Horror está em execução. Para comparar esse valor com o valor em SzEFEILE, podemos usar a função strccmp. Esta função aceita duas strings e, se elas corresponderem, retorna 0:

```
De {
if(wcscmp(pe32. szExeFile, L"Quake3-UrT.exe")==0){}


}
```

7.1.5 Mangueira de processo

Quando essas strings são combinadas, sabemos que o pe32.th32ProcessID deve conter o identificador do processo para a instância de execução do Urban Terroor. Podemos passar esse valor para OpenProcess, como fizemos nos capítulos anteriores:

```
Processo de mangue = Processo aberto (Process_All_Accesso, verdade, pe32.th32Processo);
```

7.1.6 Memória alocada

Em seguida, precisamos alocar memória dentro do Urban Terrol para armazenar o caminho completo do DLL. Para isso, usaremos o Virtuallocex, que é definido como:

```
LPVOID Virtuallocex(
Processe HPROSS,
Endereço lpvoid lp,
SIZE_T dwSize,
DWORD Flash, DWORD Flash

);
```

Por meio dos parâmetros, hProcess será o cabo de processo que obtivemos da OpenProcess. lpAddress ficará vazio porque não nos importamos onde o endereço é atribuído. dwSize será o comprimento do caminho até o nosso DLL. Como queremos alocar a memória e torná-la disponível, selecionaremos MEM_COMMIT como tipo de alocação. Por fim, como queremos gravar a memória alocada, atribuiremos proteção como página_Readwrite.

O Virtuallocex retornará um ponteiro void contendo o endereço para a memória alocada. Como precisamos desse valor na próxima vez que chamarmos o WriteProcessMemory, temos que criar uma variável para ele. Também precisamos criar uma variável para o caminho completo do DLL. Como C++ explica o método de slash inverso, precisamos usar dois para cada slash inverso. Depois de concluir todos esses parâmetros, podemos adicionar o seguinte código:

```
const char*DL_Path="C:\\Users\\\IEUser\\\source\\repos\WallHaack\Debug\
WallHaack\WallHaack\dl";
...
```

```
void*lpbazadres = Virtuallock (dll_path)+1, MEM_comando, página_readwrite);
```

7.1.7 Escreva o nome do DLL

Agora que a memória foi alocada, podemos usar o WriteProcessMemory para escrever o nome DLL na memória do Urban Terrol. O endereço escrito será o endereço que recuperamos do Virtuallocex:

```
WriteProcessMemory (process, lpbazadres, DL_Path, Strenn(DL_Path)+1, NULL);
```

7.1.8 Criar threads

Depois de gravar o caminho do DLL na memória do jogo, podemos criar um thread para executar o loadlibriary para carregar o DLL no jogo. Usaremos o Createremotthread para criar threads, mas primeiro precisamos obter o endereço da Loadlibrearya.

Laoadlibrearya existe em kernel32.dll. A Windows é responsável por carregar esse DLL em qualquer API que exija que o kernel32.dll seja incluído. Para obter o endereço de Loadlibrearya, podemos usar GetProcAddress. Esta API requer um cabo que contenha uma função DLL, neste caso kernel32.dll. Podemos obter este cabo usando GetModulehandel:

```
HMODULE KENEL32Base=GETMODULE (L "KENEL32.DLL");
```

Agora podemos carregar nosso DLL usando o CreateEMOTETHEAD. A definição do Creator EMOTETHEAD é seguinte:

```
Createremottehread (
Processe HPROSS,
lpsecurit_atributos lpthreadatos, SIZE_T
dwStackSize,
lpthead_Start_Routine lptheadres, LPVOID
lptheadres,
DWORD DWCREAGS, LPDWORD LPTHREAID

);
```

Vamos entender gradualmente cada parâmetro necessário. Este processo será o cabo de processo do Urban Terrol, o mesmo que o WriteProcessMemory. Não precisamos dos próximos dois parâmetros, então podemos passar NULL e 0 para eles. Nosso endereço inicial será o endereço de Loadlibrearya que recuperamos através da GetProcAddress. Por fim, precisamos passar um parâmetro para o loadlibrearya, que é o nosso caminho DLL, que conhecemos em chamadas para o Virtuallocex. Para o propósito da nossa seringa, também podemos ignorar os dois últimos parâmetros. Depois de fazer tudo isso, nosso telefone acabou parecendo:

```
HANDLE thread = "process, NULL, 0, (lpthead_Start_Roadline) GetProadline
(kernel32base, "Loadline"), lptheadline, 0, NULL);
```

Precisamos fazer algumas operações adicionais no thread, então salvaremos um cabo para o thread. Antes de sair, queremos que o injetor espere que o thread seja criado e concluído a execução. Podemos fazer isso através do WaitForSingleObject e do Getexits Codethread:

```
WaitForsingleock (thread, ilimitado);
GetExitCodeThread(thread, &exitCode) ;
```

7.1.9 Limpeza

Por fim, podemos liberar a memória alocada e desativar o cabo aberto após injetar o DLL:

```
Virtualfreex (process, lpbazadres, 0, MEM_RELEASE); Feche
o cabo (thread);
Feche o cabo (processo
); descanso;
```

A última interrupção sai do loop que criamos para escanear cada processo.

Tudo isso está pronto, podemos começar a City Horror, entrar no jogo e então executar nossa seringa. Se tudo correr bem, os jogadores começarão a aparecer através da parede, indicando que nosso DLL foi injetado. Se falhar, certifique-se de executar a seringa com permissões do administrador.

O código completo da seringa é mostrado no Apêndice A.

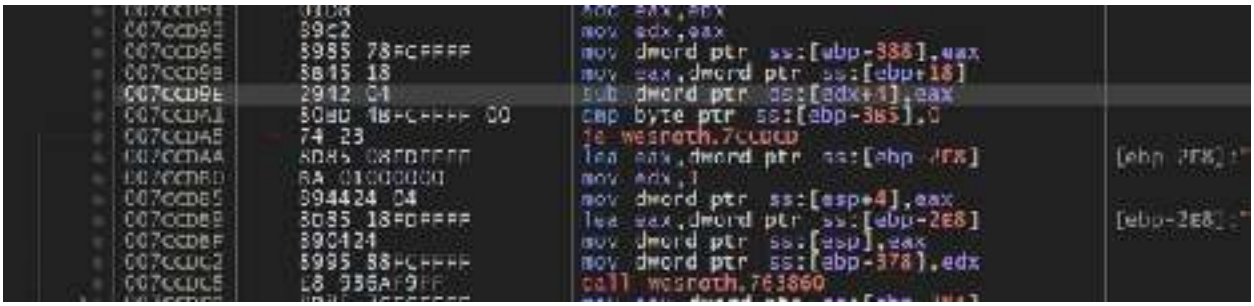
7.2 Scanner de padrão

7.2.1 Metas

Nossos objetivos neste capítulo são Wesnoth 1.14.9 e Wesnoth 1.14.12.

7.2.2 Visão Geral

No Capítulo 2.3, encontramos um subcomando responsável por deduzir moedas de ouro dos jogadores quando recrutamos uma unidade. Na versão 1.14.9 do jogo, encontramos este comando em 0x7cd9e:

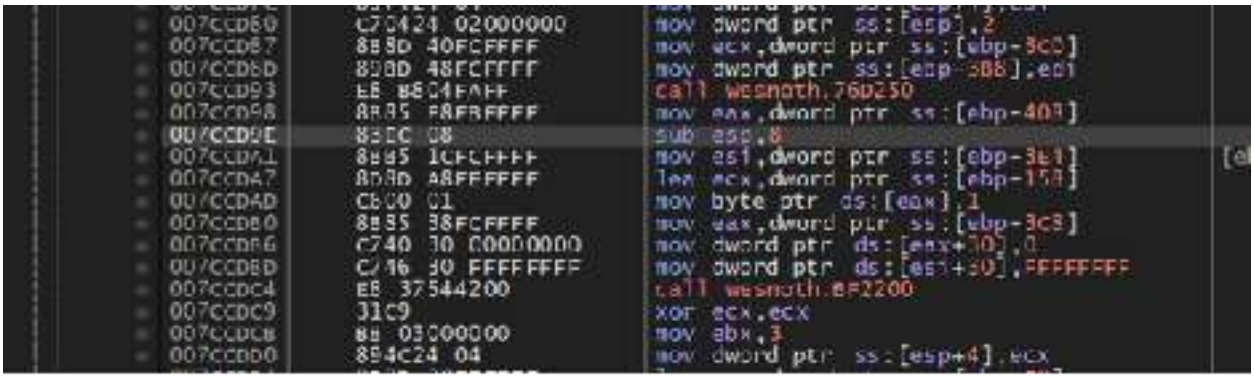


Se o código do jogo não for carregado dinamicamente, o endereço da instrução não mudará. Portanto, podemos usá-los consistentemente durante a programação. Usamos esse comportamento para construir cavernas de código em vários objetivos, como no Capítulo 3.4.

No entanto, novas versões do Wesnoth foram lançadas, como 1.14.12. Esta versão pode ser instalada usando Chocolate, assim como nossa versão de instalação 1.14.9:

```
choco Instalação WESNOTH-Version=1.14.12-y
```

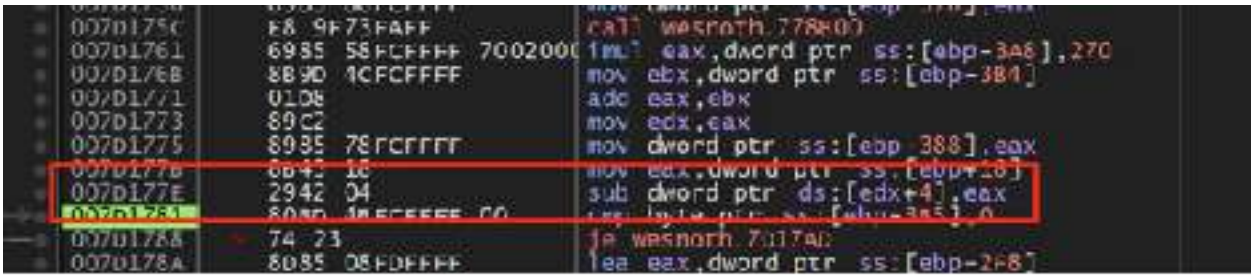
A maioria dos jogos continua lançando versões adicionais e exige atualizações antes de poder continuar jogando em servidores multijogador. Se verificarmos o 0x7cd9e no Wesnoth 1.14.12, veremos que o subcomando não existe mais:



Quando os desenvolvedores introduzem novos recursos ou corrigem bugs em cada versão, eles modificam o código do jogo. Eles então compilam essas mudanças para gerar um novo arquivo executável para o jogo. Como esse novo arquivo executável tem um código diferente, a posição de todo o código no jogo mudará. É por isso que subcomandos não aparecem mais em 0x7cd9e na versão 1.14.12.

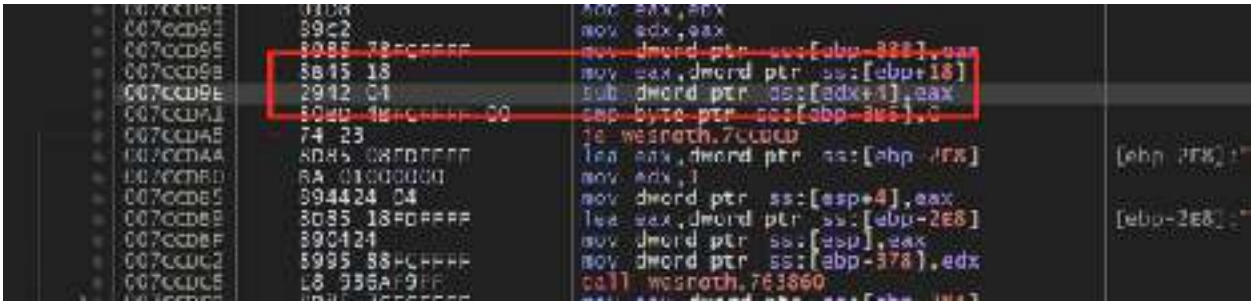
7.2.3 Código de operação

Se quisermos encontrar um novo endereço para uma subinstrução, uma maneira é repetir exatamente o mesmo método que usamos no Capítulo 2.3. Se fizermos isso, podemos reconhecer que o subcomando em 1.14.12 está localizado em 0x7D177E:



No entanto, se quisermos atualizar nossos hackers para uma versão mais recente, como 1 .14.15, teremos que repetir esse processo novamente. Este é um processo demorado, especialmente para tarefas mais complexas, como localizar o ponteiro básico do jogador.

No Capítulo 1.1, discutimos o código de operação. Cada código de operação representa a instrução a ser executada. x64dbg exibe o código de operação de cada instrução na coluna à esquerda da instrução:

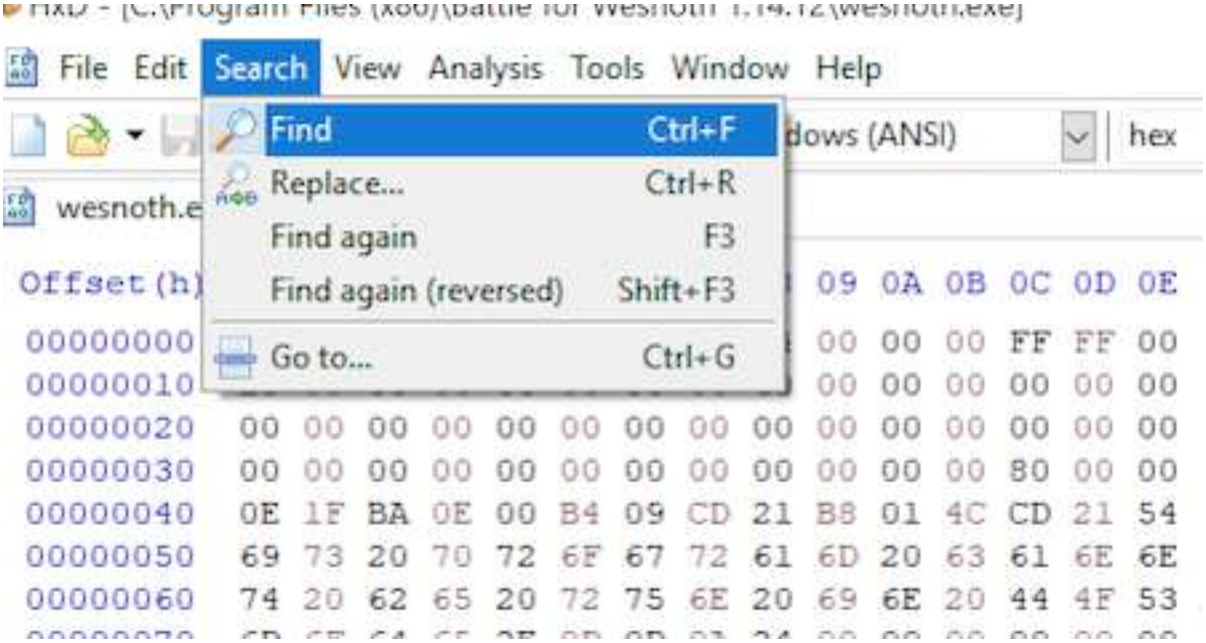


Por exemplo, o código de operação do subcomando que reconhecemos é 0x2942 04.

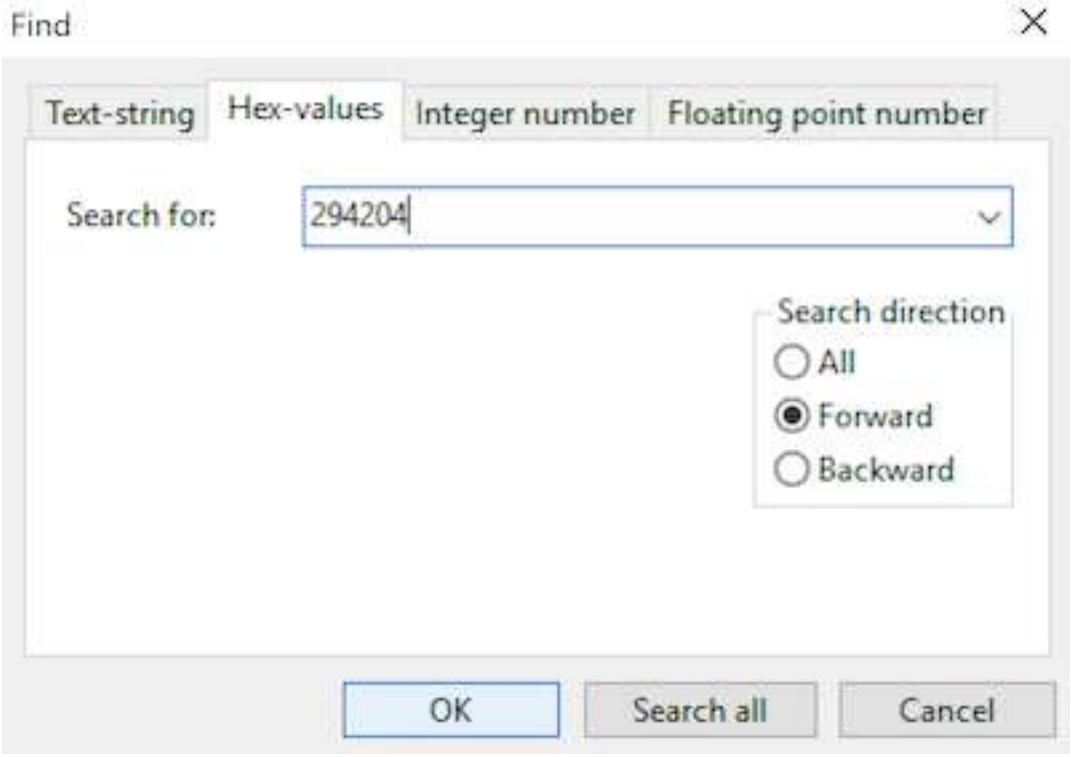
Arquivos executáveis não armazenam seu código como instruções de montagem. Em vez disso, eles o armazenam como um código de operação. A desmontagem observada em x64dbg é reconstruída com base nesses códigos de operação. Podemos verificar isso abrindo wesnoth.exe em um editor hexadecimal, um programa que exibe hexabytes do arquivo selecionado. Neste capítulo, usaremos HxD:

Instalação do Vale de Inseto e Fogo hxd

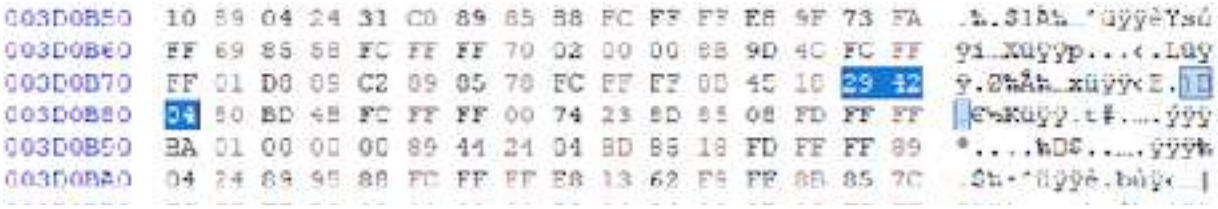
Após abrir o wesnoth.exe, podemos pesquisar o código de operação identificado acima pesquisando -> Search:



Na caixa de diálogo que aparece, queremos pesquisar o valor hexadecimal do código de operação:



A pesquisa deve destacar o valor do código de operação:



Observando valores próximos ao valor destacado, podemos ver que eles representam outros códigos de operação próximos ao subcomando.

7.2.4 Digitalização

Se compararmos os códigos de operação das subinstruções entre 1,14.9 e 1,14.12, podemos ver que eles são os mesmos. Isso ocorre porque os códigos de operação para instruções específicas sempre serão os mesmos. Como esses códigos de operação não mudam, podemos escanear esses bytes para localizar as instruções com as quais nos importamos. Isso é chamado de varredura de padrão.

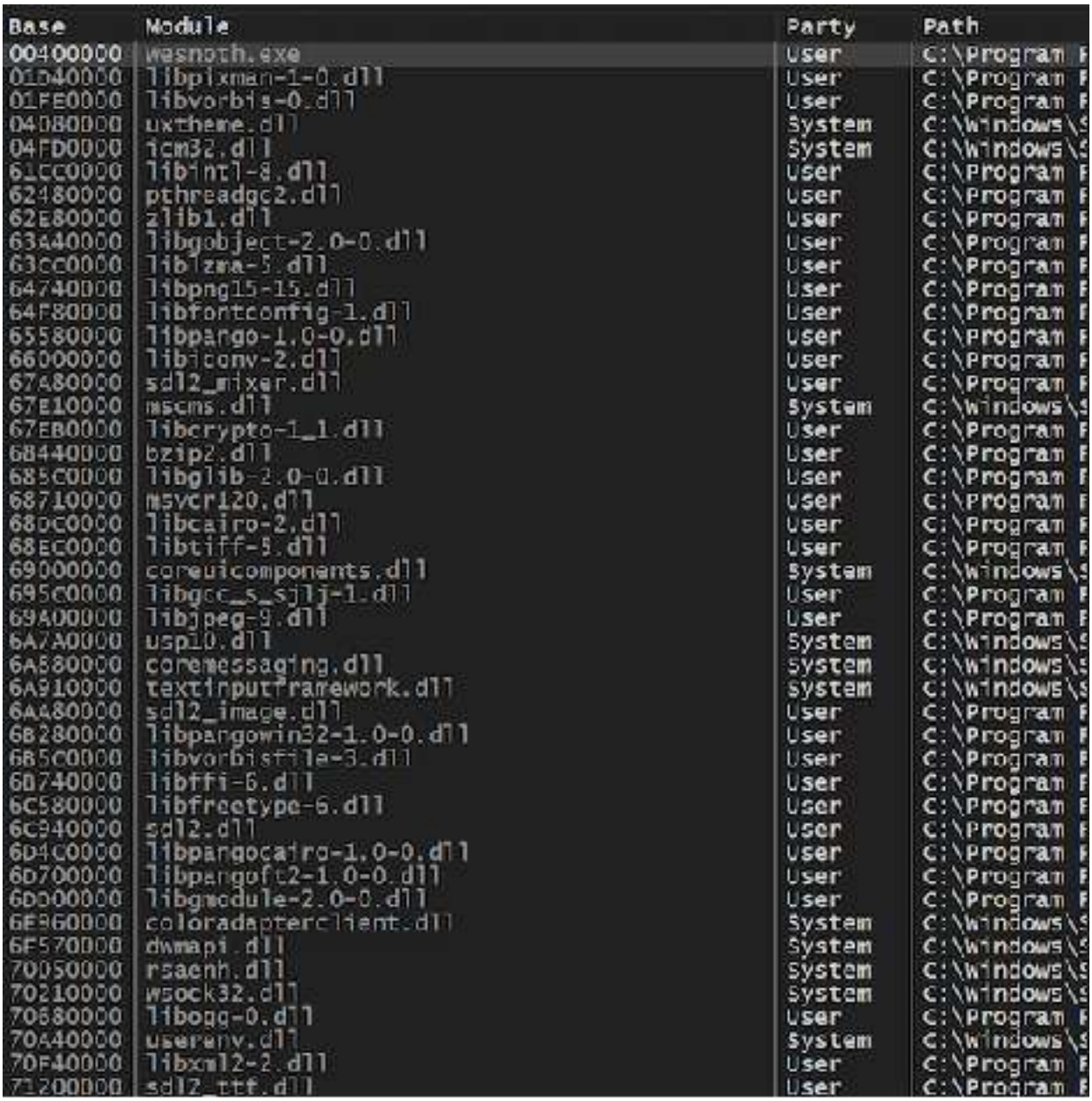
Devido à maneira como a Windows carrega arquivos PE na memória virtual, os endereços de instrução entre o editor hexadecimal e o x64dbg são diferentes. Como queremos localizar e alterar o código em execução, estamos interessados em identificar o último endereço.

Para fazer isso, precisamos ler a memória de uma instância em execução do Wesnoth e pesquisar por uma série de bytes nessa memória. Neste capítulo, escreveremos um programa externo para demonstrar esse conceito, mas o mesmo comportamento pode ser usado dentro do DLL para atualizar automaticamente o deslocamento.

Como queremos localizar um processo em execução e recuperar um cabo de processo , podemos começar com os fundamentos que discutimos no Capítulo 7.1:

```
#include<windows.h>
#include<tlhelp32.h>
#Incluindo <stdio.h>

int main(int argc, char**argv){
	Instantâneo do cabo = 0;
	processEntry32 pe 32={0};

	pe32. dwSize = dwSize (PROCESSENTRY32);
	snapshot=Creatotolhelp32Snapshot (th32cs_Snapshot, 0);
	process 32 first (instantâneo, &P32);

	De {
if(wcscmp(pe32.sconectividade, L"wesnoth.exe") ==0){HANDLE process=
Open process (process_al_acesso, verdade, pe32.th32 process);


Feche o cabo (processo);
		descanso;
}
	}while (Process32Next (instantâneo, &pe32
)); Retornar a 0;

}
```

Com o cabo de processo do Wesnoth, nosso próximo objetivo é ler a memória do processo em um buffer que podemos escanear. Entretanto, um processo consiste em muitos módulos ou snippets de código. Por exemplo, o processo Wesnoth tem módulos para o código do jogo principal (wesnoth.exe), código comprimido (zlib1.dll) e código gráfico (sdl.dll). Podemos usar a aba Symbol do x64dbg para observar todos os módulos carregados no processo:



Base	Module	Party	Path
00400000	wesnoth.exe	User	C:\Program F
01D40000	libpixman-1-0.dll	User	C:\Program F
01FE0000	libvorbis-0.dll	User	C:\Program F
04080000	uxtheme.dll	System	C:\Windows\S
04FD0000	icm32.dll	System	C:\Windows\S
61CC0000	libintl-8.dll	User	C:\Program F
62480000	pthreadgc2.dll	User	C:\Program F
62E80000	zlib1.dll	User	C:\Program F
63640000	libgobject-2.0-0.dll	User	C:\Program F
63CC0000	liblzma-5.dll	User	C:\Program F
64740000	libpng15-15.dll	User	C:\Program F
64F80000	libfontconfig-1.dll	User	C:\Program F
65580000	libpango-1.0-0.dll	User	C:\Program F
66000000	libiconv-2.dll	User	C:\Program F
67480000	sdl2_mixer.dll	User	C:\Program F
67E10000	mscms.dll	System	C:\Windows\S
67EB0000	libcrypto-1_1.dll	User	C:\Program F
68440000	bzip2.dll	User	C:\Program F
685C0000	libglib-2.0-0.dll	User	C:\Program F
68710000	msvcr120.dll	User	C:\Program F
68DC0000	libcairo-2.dll	User	C:\Program F
68EC0000	libtiff-5.dll	User	C:\Program F
69000000	coreuicomponents.dll	System	C:\Windows\S
695C0000	libgcc_s_sjlj-1.dll	User	C:\Program F
69A00000	libjpeg-9.dll	User	C:\Program F
6A7A0000	usp10.dll	System	C:\Windows\S
6A880000	coremessaging.dll	System	C:\Windows\S
6A910000	textinputframework.dll	System	C:\Windows\S
6A480000	sdl2_image.dll	User	C:\Program F
68280000	libpangowin32-1.0-0.dll	User	C:\Program F
685C0000	libvorbisfile-3.dll	User	C:\Program F
6D740000	libffi-6.dll	User	C:\Program F
6C580000	libfreetype-6.dll	User	C:\Program F
6C940000	sdl2.dll	User	C:\Program F
6D4C0000	libpangocairo-1.0-0.dll	User	C:\Program F
6D700000	libpangoft2-1.0-0.dll	User	C:\Program F
6DD00000	libgmodule-2.0-0.dll	User	C:\Program F
6E960000	coloradapterclient.dll	System	C:\Windows\S
6F570000	dwmapi.dll	System	C:\Windows\S
70050000	rsaenh.dll	System	C:\Windows\S
70210000	wsock32.dll	System	C:\Windows\S
70680000	libogg-0.dll	User	C:\Program F
70A40000	userenv.dll	System	C:\Windows\S
70F40000	libxml2-2.dll	User	C:\Program F
71200000	sdl2_ttf.dll	User	C:\Program F

Como nossas subinstruções estão no módulo wesnoth.exe, queremos apenas escanear essa memória. Para isso, queremos identificar o local do módulo e seu tamanho. A API Snapshot CreateToolhelp32Snapshot também nos permite usar módulos para o Module32First e o Module32Next iterativo:

```
if(wcscmp(pe32.sconectividade, L"wesnoth.exe") ==0){HANDLE
process=Open process (process_al_acesso, verdade, pe32.th32
process);

module_snapshot=0;
```

```
Entrada do módulo 32 ME32;

me32 2. dwSize = dwSize (moduleentry 32);
module_snapshot=Creatotolhelp32Snapshot (th32cs_Snapshot, pe32.th
32ProcessID);
Module32First (module_snapshot, &me32);
De {
      if(wcscmp(me32.szmodul, L"wesnoth.exe") ==0){

descanso;
}
} while(module_snapshot,&me32);

Feche o cabo (processo);
      descanso;
}
```

Neste ponto, a estrutura me32 salvará vários membros com os quais nos importamos: o endereço base do módulo, modbasesaddr, e o tamanho do módulo, modbaseisize. Usaremos esses dois membros para alocar um buffer e ler a memória do módulo no buffer:

```
unsigned char*buffer=(unsigned char*)Calc (1, me32. Modbassize);
DWORD bytes_read=0;

ReadProcessmesy (process, (void*) me32.modbasidr, buffer, me32.modbasidar, &
bytes_read);

//Código de digitalização
gratuito (buffer);
```

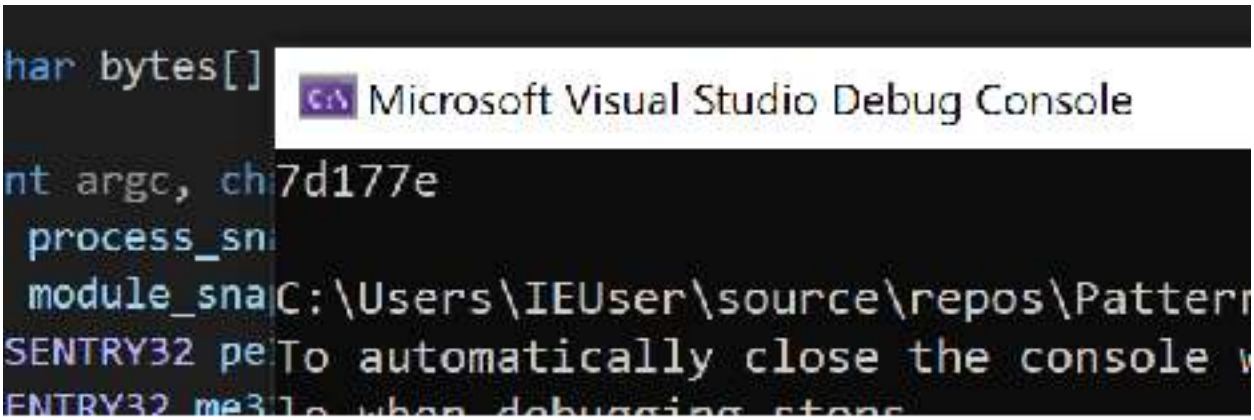
Neste ponto, nosso buffer contém conteúdo de memória desde a base do módulo wesnoth.exe até o final do módulo. Esta memória contém o código de operação do código do jogo. Agora podemos escanear essa memória para encontrar nosso padrão de byte.

Para cada byte no buffer, verificaremos se o padrão que começa com esse byte existe. Caso contrário, continuaremos para o próximo byte. Se todos os bytes forem combinados, combinaremos o deslocamento no buffer de impressão da biblioteca do módulo wesnoth.exe:

```
Bytes de caracteres não símbolos []={0x29,0x42,0x04}
```

```
for (unsigned int i=0; i<me32 2. modbasezize-sizef(bytes);i++){
for(int j=0; j<sizeof(bytes); j++){
if(bytes[j]!=Buffer[i+j]){
descanso;
}

if(j+1==sizeof(bites)){printf("%x\n", i+(DWORD)me32.
modbasidr);}

}}
```

Se iniciarmos o Wesnoth 1.14.12 e executarmos nosso scanner, ele imprimirá corretamente a localização do subcomando:



Podemos usá-lo em qualquer versão do Wesnoth para localizar subinstruções com as quais nos importamos. Os códigos completos deste capítulo podem ser comparados no Apêndice A.

7.3 scanner de memória

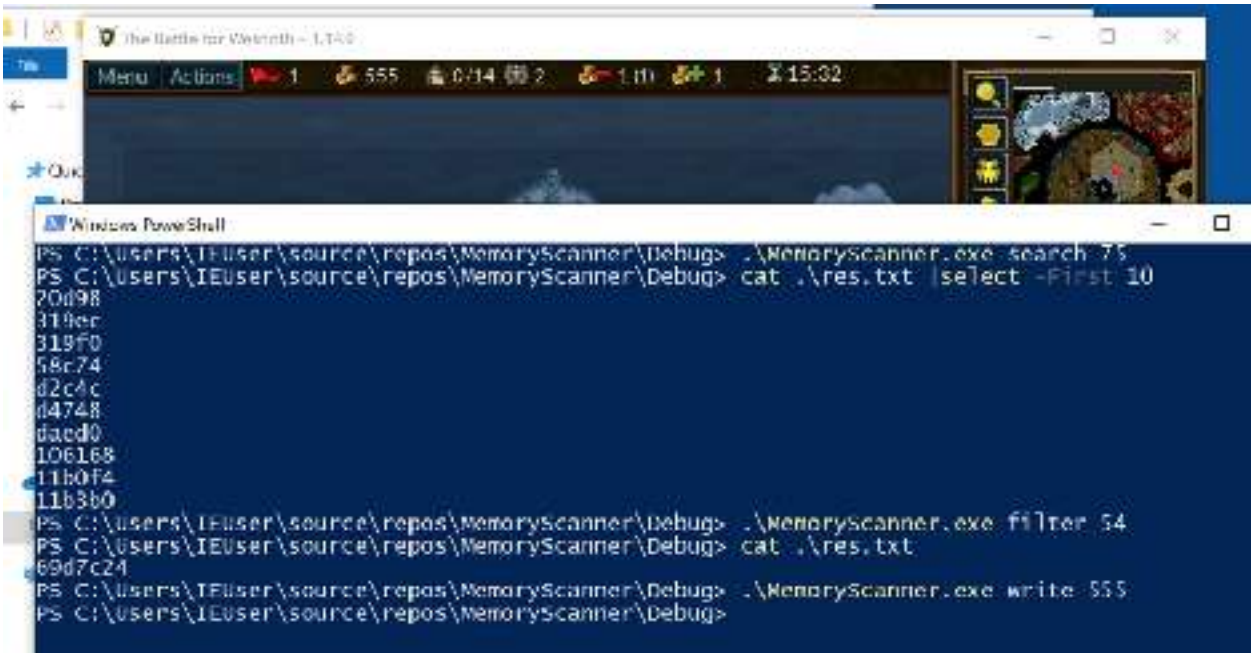
7.3.1 Objetivos

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

7.3.2 Visão Geral

No capítulo anterior, usamos mecanismos de fraude para pesquisar endereços de memória e alterar seus valores. O mecanismo de fraude é um programa chamado scanner de memória. Um scanner de memória permite que você pesquise e edite a memória dentro de um processo.

Nosso objetivo neste capítulo é criar um scanner de memória que manipule o valor DWORD do jogo Wesnoth.



7.3.3 Compreensão

Um scanner de memória tem três operações principais:

1. Pesquise por um valor em toda a memória.
2. Filtre endereços identificados anteriormente com base no novo valor.
3. Defina o endereço de memória para um determinado valor.

No capítulo anterior, criamos um scanner de padrões que pesquisaria uma série de bytes no módulo Wesnoth primário. Podemos usar a mesma técnica para pesquisar valores na memória. Entretanto, neste caso, escanearemos toda a memória de 0x 00000000 a 0x7fffffff. Este intervalo de endereço representa todo o espaço de endereço virtual que um arquivo Windows executável de 32 bits pode acessar. Ao escanear, salvaremos qualquer endereço definido para um determinado valor.

Para filtrar esses endereços, realizaremos as mesmas operações de digitalização descritas acima, mas com uma grande diferença: escanearemos apenas os endereços salvos identificados na etapa de digitalização anterior, em vez de 0x00000000 para 0x7fffffff. Qualquer endereço que ainda corresponda ao valor fornecido será salvo novamente. Dessa forma, podemos continuar filtrando a lista de endereços válidos.

Por fim, para escrever o endereço, podemos usar a mesma tecnologia WriteProcessMemory mencionada no Capítulo 3.2.

7.3.4 Estrutura do Programa

Antes de escrever um programa, precisamos determinar como lidar com várias operações e como transferir dados de uma operação para outra.

Como nossos scanners de memória têm três operações diferentes a serem executadas, precisamos determinar como lidar com essas situações. Uma abordagem é criar um programa separado para cada operação e então transferir dados entre os três programas. Entretanto, esse método exige que copiemos a lógica entre vários programas, como a lógica para abrir o cabo do processo.
Neste capítulo, usaremos parâmetros de linha de comando para especificar as operações que queremos executar. Por exemplo, se quisermos pesquisar o valor de 50, chamaremos nosso programa da seguinte forma:

```
MEMORYScanner.exePesquisa 50
```

Como precisamos chamar o scanner várias vezes, precisamos de uma maneira de passar os resultados de uma operação para a outra. A maneira mais fácil de fazer isso é usar arquivos. Por exemplo, se pesquisarmos um valor, o arquivo preencherá todos os endereços que correspondem ao valor.

Valor. Quando filtramos, leemos o endereço do arquivo e, se ainda corresponderem, colocamos o novo endereço no arquivo.

7.3.5 Mangueira de processo

Para ler e escrever memória a partir do Wesnoth, precisamos de um cabo de processo. Vamos fazer isso usando a mesma abordagem que no capítulo anterior:

```
#include<windows.h>
#include<tlhelp32.h>
#Incluindo <stdio.h>

int main(int argc, char**argv){
      Maneira de frase process_snapshot=0;
      processEntry32 pe 32={0};

      pe32. dwSize = dwSize (PROCESSENTRY32);

      process_snapshot=Creatotolhelp32snapshot (th32cs_snapshot, 0);
      Process32First(process_snapshot, &pe32);

      De {
            Se(wcscmp(pe32. szExeFile, L"wesnoth.exe") ==0){HANDLE process
                  =Open process(PROCESS_ALL_ACCESS, verdade,
PE32.Th32Procesid);

//Operações de processamento

                  Feche o cabo (processo);
                  descanso;
            }
      }while (process_snapshot, &pe32); Retornar a 0;


}
```

Passaremos esse processo para todas as nossas operações.

7.3.6 Operação

Em seguida, podemos adicionar nossas operações. Para acessar os parâmetros da linha de comando passados ao nosso programa, podemos usar os parâmetros argv. argv[0] sempre salvará o nome do nosso programa na linha de comando (MemoryScanner.exe), e argv[1] representa o primeiro parâmetro.

Todos os parâmetros são passados na forma de string. Queremos que nosso programa pesquise o valor DWORD. Para converter uma string em um valor que pode ser usado para pesquisar DWORD, usaremos strtol ou (str)(to)(l)ong:

```
//Operações de processamento
char*p;
Valor comprimento = strtol (argv[2],&p, 10);

Se(strcp(argv[1], "pesquisar")==0){
pesquisar (processe, valor);
}
else if(strcp(argv[1], "filtro") ==0){
filtro (profil, valor);
}
Esse if(strcp(argv[1], "write")==0){write(
profile, valor);
}
```

Com a base, agora podemos escrever cada uma dessas funções.

7.3.7 Pesquisa

Começaremos com a função de pesquisa:

```
void search (const HANDLE process, const int passed_val){
```

Como discutimos na seção anterior, armazenaremos os resultados da pesquisa em um arquivo de texto. Como ~~fizemos no C~~apítulo 6.4, usaremos fopen_s para criar um arquivo de texto que pode ser escrito:

```
file*temp_filo=NULL;
fopen_s(&temp_filo, "res.txt", "w");
```

Como sabemos, a memória não tem uma estrutura específica. Por exemplo, memórias de 0x12345678 a 0x1234567C podem armazenar valores de 0x44 0x45 0x41 0x45 0x45. Se lida como DWORD, a memória salva o valor de 1145389381. Entretanto, se cada byte for lido como um caractere, a memória manterá esse valor. Em scanners de memória como o Cheat Engine, você pode selecionar o tipo de dados que deseja escanear. Neste capítulo, escanearemos toda a memória como se fosse um DWORD. Isso nos permitirá pesquisar por valores numéricos, como gold.
Nossa operação de pesquisa escaneia toda a memória de 0x00000000 a 0x7fffffff e compara cada 4 bytes com o valor passado no segundo parâmetro. Anteriormente, usávamos ReadProcessMemory para ler um único DWORD de 4 bytes. No entanto, o ReadProcessMemory nos permite ler memória de qualquer tamanho em qualquer tipo de buffer de alocação.
Embora o Wesnoth possa usar toda a memória de 0x00000000 a 0x7FFFFF, ele primeiro precisa ser acessado por meio de várias solicitações de API, como Virtualloc. Se o Wesnoth não solicitar acesso a um pedaço de memória, ele não poderá ler ou gravar dados para ele. Usamos o cabo do Wesnoth para ler a memória, então precisamos considerar esse comportamento.

Se tentarmos ler toda a memória de 0x00000000 a 0x7fffffff usando uma chamada ReadProcessMemory, a chamada falhará. Isso ocorre porque o comportamento do ReadProcessMemory é que, se encontrarmos um pedaço de memória que não podemos acessar, ele falha imediatamente e coloca um valor null no buffer. Portanto, precisamos dividir a solicitação de leitura em blocos. Dessa forma, se tentarmos escanear um bloco que não foi alocado no Wesnoth, a leitura do único bloco falhará.

Podemos escolher qualquer valor para o tamanho do nosso bloco, mas há um compensação entre velocidade e precisão. Quanto maior cada bloco, mais rápido será o processo de varredura, mas como parte do bloco não é acessível, mais áreas de memória podem não ser lidas com sucesso. Neste capítulo, selecionaremos o tamanho do bloco de 2056 ou 0x808:

```
#Definição de tamanho 0x00000808
```

Em seguida, atribuiremos um buffer que pode acomodar dados do tamanho de bloco:

```
unsigned char*buffer=(unsigned char*)Calc (1, tamanho);
```

Em seguida, passamos por cada bloco de memória de 0x00000000 a 0x7fffffff em loop e leemos o bloco no buffer:

```
DWORD bytes_read=0;

for (DWORD i=0x000000; i<0x7fffffff;i+=tamanho) {process, (vod*)i,
      buffer, tamanho, &bytes_read);
```

Por fim, convertemos cada 4 bytes do buffer em um DWORD e determinamos se seu valor é igual ao parâmetro passado. Se for esse o caso, gravaremos sua localização no arquivo de resultados:

```
for (int j=0; j< tamanho-4; j+=4) {DWORD val=0
;
MEMCY (&val, &buffer[j], 4);
Se(val==passed_val){fprintf(temp_fil, "%x\
n", i+j);}

}
```

Se a leitura falhar, nosso buffer conterá apenas 0, e esta última etapa será inútil. Após concluir o loop ReadProcessMemory, fecharemos o arquivo e liberaremos a memória do buffer:

```
fclose (temp_filme
); Livre (tampão);
```

Nossa função de pesquisa já está pronta. Se você criar este código e pesquisar o valor gold no Wesnoth, verá que res.txt agora contém vários endereços:

```
Pesquisa MemoryScanner.exe 75
```

7.3.8 Filtragem

A próxima ação na qual vamos nos concentrar é a filtragem. A operação de filtragem obtém uma lista de endereços gerados pela operação de pesquisa e verifica se esses endereços são iguais aos novos valores. Se o endereço for realmente igual a esse valor, ele será salvo. Caso contrário, exclua:

```
void filter (const HANDLE process, const int passed_val){
```

Executaremos a operação de filtragem em duas partes:

1. Leia cada endereço de memória de res.txt e, se corresponder ao novo valor, salve-o em res_fil.txt.
2. Copie res_fil.txt para res.txt e exclua res_fil.txt.

O resultado final será um novo arquivo res.txt que contém apenas endereços filtrados. Este modelo nos permitirá filtrar várias vezes. Primeiro, abriremos res.txt para ler (r), abriremos res_fil.txt para gravar (w):

```
file*temp_filo=NULL;
file*temp_file_file_filter=NULL;
fopen_s(&temp_filo, "res.txt", "r");
fopen_s(&temp_file_filtro, "res_file.txt", "w");
```

Em seguida, leremos cada endereço em res.txt linha por linha e
Esse endereço. Se o valor corresponder aos nossos parâmetros, escreveremos o endereço em res_fil.txt:

```
Endereço DWORD = 0;
while(fscanf_s(temp_file, "%x\n",&adress)! =EOF){
DWORD val=0;
DWORD bytes_read=0;

ReadProcessMemory (processo, (void*) endereço, &val, 4, &bytes_read); Se(
val==passed_val){
fprintf(temp_file_filter, "%x\n", endereço);}

}
```

Depois que todos os endereços filtrados estiverem incluídos em Ress_fil.txt, desativaremos Ress_fil.txt e Ress_fil.txt. Em seguida, abriremos esses arquivos na ordem oposta, gravaremos usando res.txt e leremos usando res_fi.txt:

```
fclose (temp_filme);
fclose (temp_file_filter);

fopen_s(&temp_filo, "res.txt", "w");
fopen_s(&temp_file_filtro, "res_file.txt", "r");
```

Em seguida, passaremos por cada endereço em ress_fil.txt e copiaremos para ress.txt:

```
while(fscanf_s(temp_file_filtro, "%x\n", &adres)!=EOF){fprintf
(temp_filtro, "%x\n", adres);
}
```

Como res.txt agora contém nosso endereço, fecharemos cada arquivo e excluiremos res_fil.txt:

```
fclose (temp_filme);
fclose (temp_file_filter
); remove ("res_fil.txt");
```

Agora podemos pesquisar e filtrar endereços. Se você pesquisar seu ouro no Wesnoth, comprar uma unidade e filtrar o valor do seu ouro, você deve ter apenas um valor restante. Se você ativar o mecanismo de fraude e repetir essas etapas, poderá verificar se o endereço que você identificou corresponde ao endereço do mecanismo de fraude. Isso mostra que nosso scanner encontrou corretamente o endereço de memória.

```
Filtro MemoryScanner.exe 54
```

7.3.9 Escrita

A última operação importante de um scanner de memória é gravar o valor em um endereço de memória reconhecido:

```
void write (const HANDLE process, const int passed_val){
```

Esta operação é a mesma que usamos no Capítulo 3.2. Para cada endereço em res.txt, usaremos o WriteProcessMemory para escrever o valor do parâmetro fornecido no endereço:

```
file*temp_filo=NULL;
fopen_s(&temp_filo, "res.txt", "r");

Endereço DWORD = 0;
while(fscanf_s(temp_file, "%x\n", &adss)! =EOF){DWORD
bytes_written=0;
```

```
WriteProcessMemotory(Processo, (void*)Adress, &passed_val, 4, &
BYTES_Writeten);
}

fclose(temp_filme);
```

Com este código, agora podemos gravar qualquer valor que desejamos nos endereços que pesquisamos e filtramos anteriormente:

```
memoryscanner.exeEscreva para 555
```

Os códigos completos deste capítulo podem ser comparados no Apêndice A.

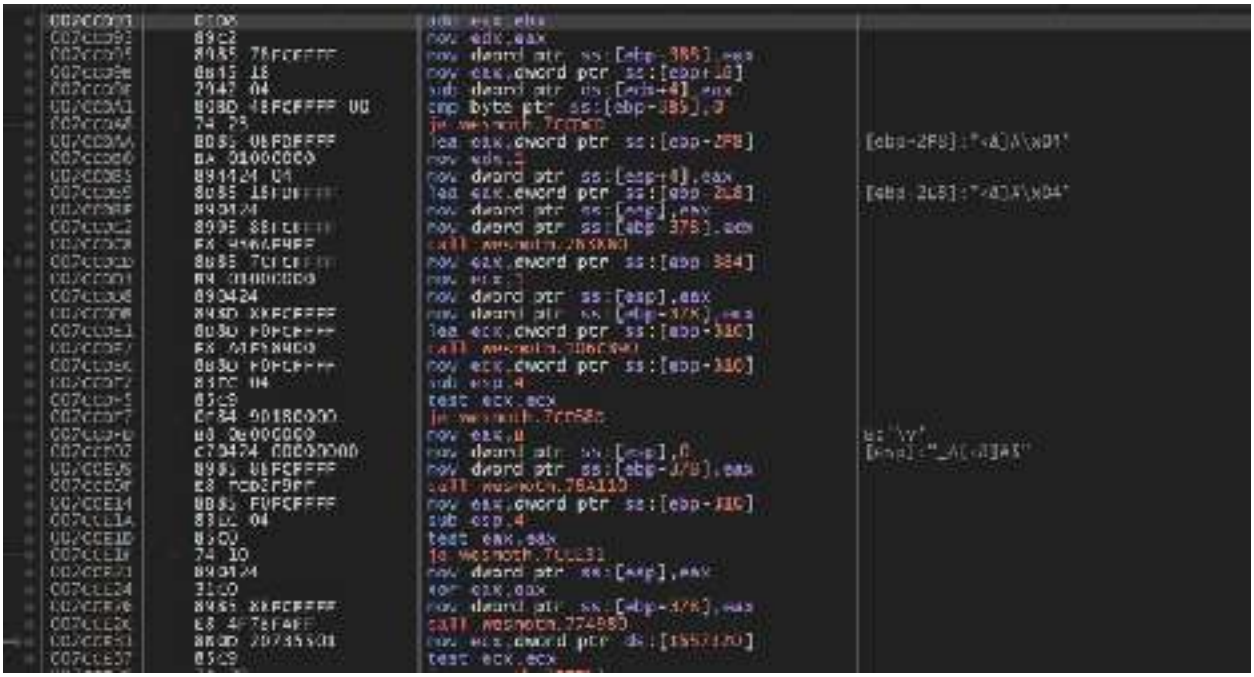
7.4 Programa de Descompilação

7.4.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

7.4.2 Visão Geral

No capítulo anterior, usamos x64dbg para depurar e reverter o jogo. Ao visualizar esses jogos no x64dbg, podemos ver as instruções que o jogo está executando. Por exemplo, vemos as seguintes instruções responsáveis por reduzir o ouro dos jogadores ao recrutar uma unidade na Wesnoth:



No Capítulo 7.2, sabemos que essas instruções são armazenadas como códigos de operação, que são valores de byte. O processo de conversão desses códigos de operação em instruções é chamado de desmontagem. Neste capítulo, explicaremos como criar um programa de descompilação limitado.

O código-fonte completo discutido neste capítulo é mostrado no Apêndice A.

7.4.3 Isenção de responsabilidade

Escrever um programa de descompilação é uma tarefa complexa que leva muito tempo. Mesmo que você ofereça suporte eficiente a um único conjunto de instruções, leva semanas para ler as especificações e implementá-las. O método apresentado aqui deve servir como ponto de partida, mas é importante ressaltar que o método não será estendido . O principal objetivo deste capítulo é explicar como esses conceitos funcionam. Para exemplos de um programa de desmontagem totalmente funcional, veja Capstone Engine.

7.4.4 Explicação

Para que a CPU entenda e execute todas as operações encontradas, essas operações devem ter um formato consistente. Cada código de operação deve receber uma instrução específica. Por exemplo, como vimos no capítulo anterior, o código de operação 0xE8 sempre representa uma instrução de chamada. Esse mapeamento de operações para instruções é chamado de conjunto de instruções do processador.

Cada CPU pode implementar um conjunto de instruções exclusivo. No entanto, a maioria dos jogos baseados em Windows foi compilada para serem executados em um processador de 32 bits baseado em Intel. Esses processadores geralmente implementam uma versão do conjunto de instruções x86. Especialmente para processadores Intel, isso é chamado de IA-32.

O conjunto de instruções x86 é complexo e envolve muitas operações diferentes. Essas operações também podem ter comprimentos diferentes. Por exemplo, na captura de tela da página acima, vemos que o comando mov na segunda linha (0x7cd93) é de 2 bytes (0x89C2), enquanto o comando mov na terceira linha é de 6 bytes (0x8985 78FCFFFF). Para que a CPU entenda o comprimento das instruções, esses dados devem ser codificados em bytes de alguma forma.

7.4.5 Referência do Conjunto de Instruções

Suponha que você queira criar um compilador que aceite o seguinte código C++ e gere um arquivo binário que pode ser executado em um processador compatível com x86:

```
int x=2;
```

Este código pode ser convertido em uma assembleia de várias maneiras, como:

```
mov[x], 2
```

o
u

```
m eax, 2 m[x
], eax
```

Já vimos que os comandos mov vêm em várias formas e têm diferentes comprimentos. Como desenvolvedores de compilador, precisamos saber qual formato usar para gerar nosso código binário.

Para resolver esse problema, empresas como a Intel emitiram referências de conjuntos de instruções. Estas contêm uma lista completa de todas as instruções comuns e seus códigos de operação relacionados, bem como outras informações de arquitetura, como como codificar o comprimento das instruções. Referências IA-32 podem ser encontradas aqui.

Quando construímos um programa de desmontagem, usamos essa referência para entender as instruções. Além disso, usaremos outra referência (aqui) para ajudar a encontrar códigos de operação desconhecidos e a qual instrução eles estão associados.

7.4.6 Código de operação do processo de dump

Assim como no Capítulo 7.2, nosso alvo neste capítulo é Vesnos. Usaremos o mesmo código deste capítulo para localizar, adicionar e ler o código de operação do jogo no buffer:

```
int main(int argc, char**argv){
      Mancha de frase process_snapshot=0;
      module_snapshot=0; processEntry32 pe
      32={0}; Entrada do módulo 32 ME32;


      Código de saída DWORD = 0;

      pe32. dwSize = dwSize (PROCESSENTRY32);
      me32 2. dwSize = dwSize (moduleentry 32);

      process_snapshot=Creatotolhelp32snapshot (th32cs_snapshot, 0);
      Process32First(process_snapshot, &pe32);

      De {
```

```
            if(wcscmp(pe32. szExeFile, L"wesnoth.exe") ==0){
                  module_snapshot=
Creatoteolhelp32Snapshot (TH32CS_SNAPMODULE, pe32.th32ProcessID);

                  Processo de cabo = Processo aberto (Process_All_Acesso, verdade,
PE32.Th32Procesid);

                  Module32First (module_snapshot, &me32);
                  De {
                        if(wcscmp(me32.szmodul, L"wesnoth.exe") ==0){unsigned
                              char*buffer=(unsigned)
char*)Caloc (1 me32. Modbassize);
                              DWORD bytes_read=0;

                              ReadProcessMemory (processo,
(void*)me32.modbasidr, buffer, me32.modbasidar, &bytes_read); //O buffer contém
o código de operação do jogo


                              Livre (tampão);
                              descanso;
                        }

                  } while(module_snapshot,&me32);

                  Feche o cabo (processo);
                  descanso;
            }
      }while(process_snapshot,&pe32);

      Retornar a 0;
}
```

Validaremos nosso programa de desmontagem no mesmo conjunto de instruções que vemos na Seção 7.4.2 (começando pelo endereço 0x7cd91). Também desmontaremos apenas instruções de 0x50 bytes. Primeiro, simplesmente depositaremos todos os códigos de operação:

```
#Definir endereço inicial 0x7cd91
...

unsigned int i=Start_Adress-(DWORD)me32.modbasidr; while(i<
Start_Adress+0x50-(DWORD)me32.modbasidr){printf("%x", buffer
[i]);
      i++;
```

```
printf("\n");}
```

Como o código de operação é lido no buffer, o código acima precisa ser deslocado para i dessa maneira. Como podemos ver no Capítulo 7.2, o módulo principal do jogo é carregado no endereço 0x400000. Entretanto, a instrução é armazenada na posição 0 no buffer. Para acessar o código de operação a partir de 0x7cd91, precisamos confirmar

0x7cd91 para 0x400000 e use esta posição no buffer. Quando executado, este código produz os seguintes resultados:

Podemos ver que esses códigos de operação são consistentes com os valores observados em x64dbg.

7.4.7 Instruções ADD

A partir de cima, vemos que o primeiro código de operação é 0x01. Dê uma olhada no nosso site de referência aqui e vemos que esta é uma instrução de adição:

pf	0f	po	so	o	proc	st	m	rl	x	mnemonic	op1	op2
		00		r					L	ADD	**r/m8**	r8
		01		r					L	ADD	**r/m16/32/64**	r16/32/64
		02		r						ADD	**r8**	r/m8

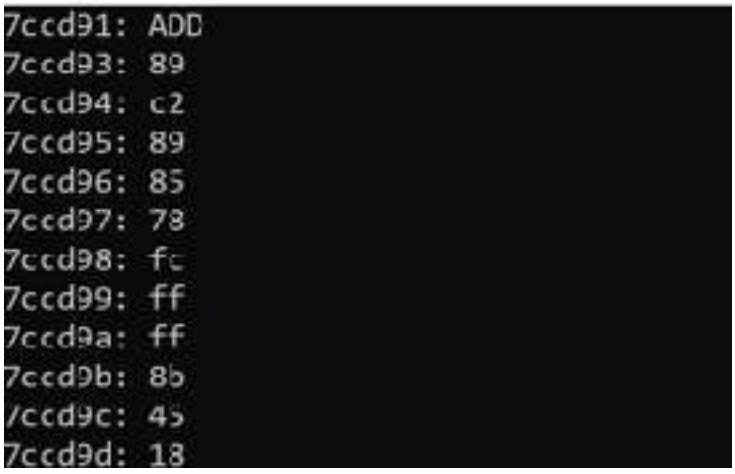
Se olharmos para a Seção 3.2 da referência, veremos que esta instrução adiciona um registro de 32 bits a um registro de 32 bits:

01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.

Ainda precisamos descobrir como esses registros são codificados, mas agora podemos modificar nosso loop principal para imprimir uma instrução add sempre que encontrarmos 0x01. Como sabemos que essa instrução é de 2 bytes, quando encontramos o próximo código de operação, também aumentamos o valor para superá-lo. Também podemos adicionar código para imprimir o endereço atual da instrução:

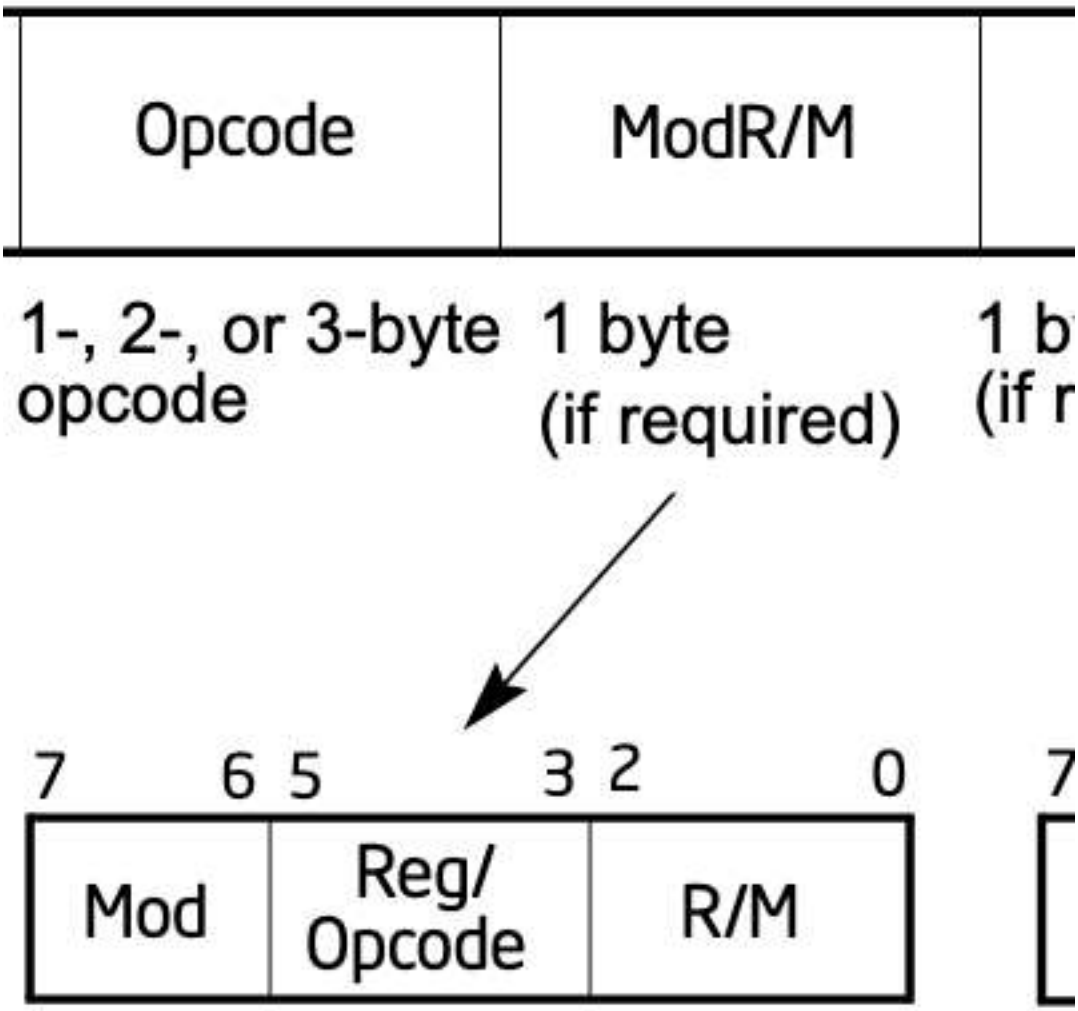
```
while(i<START_ADDRESS+0x50-(DWORD)me32.modBaseAddr){print("
%x:\t", i+(DWORD)me32.modBaseAddr); switch(FBUER[i]){

Caso 0x1:
printf("adicionar"); i+
+;
i++;
descanso;
Padrão padrão:
printf("%x", buffer[i]); i++;

Quebrar;}


printf("\n");}
```

Ao executar este código, agora veremos que a primeira instrução add está corretamente desmontada:



7.4.8 Operando de decodificação

Depois de 0x01, o próximo código de operação é 0xD8. Sabemos que este 0xD8 é responsável por codificar valores eax e ebx até certo ponto. Assim como o código de operação, o método exato para decodificar esse valor deve existir em algum lugar neste manual. Se olharmos para a Seção 2.1 do manual de referência, veremos um valor ModR/M de 1 byte após o código de operação:



Se rolarmos para baixo para a Tabela 2-2, podemos ver como esses valores são organizados:

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Encontramos o valor de 0xD8 e vemos na linha eax e na coluna ebx. Como esse valor é armazenado de forma consistente, podemos escrever uma função para recuperá-lo:

```
int Decode_Oper (unsigned char*buffer, int Locação){return 1
;

}
```

Até agora, vimos que os operandos têm 1 byte de comprimento, então retornamos o valor de 1 para aumentar o loop corretamente. Podemos chamar esta função a partir do loop principal:

```
Caso 0x1:
printf ("adicionar
"); i++;
```

```
i+=decode_Opend(buffer, i);
descanso;
```

Voltando à tabela, podemos ver que temos 8 valores possíveis: eax, ecx, edx, ebx, esp, ebp, esi e edi. Podemos colocá-los em um array de caracteres para referência em uma função:

```
const char modrm_value[8][4]={
	"eax",
	"ecx",
	"ebx",
	"esp",
	" ebp
	",   "
	esi",
	"edi"

};
```

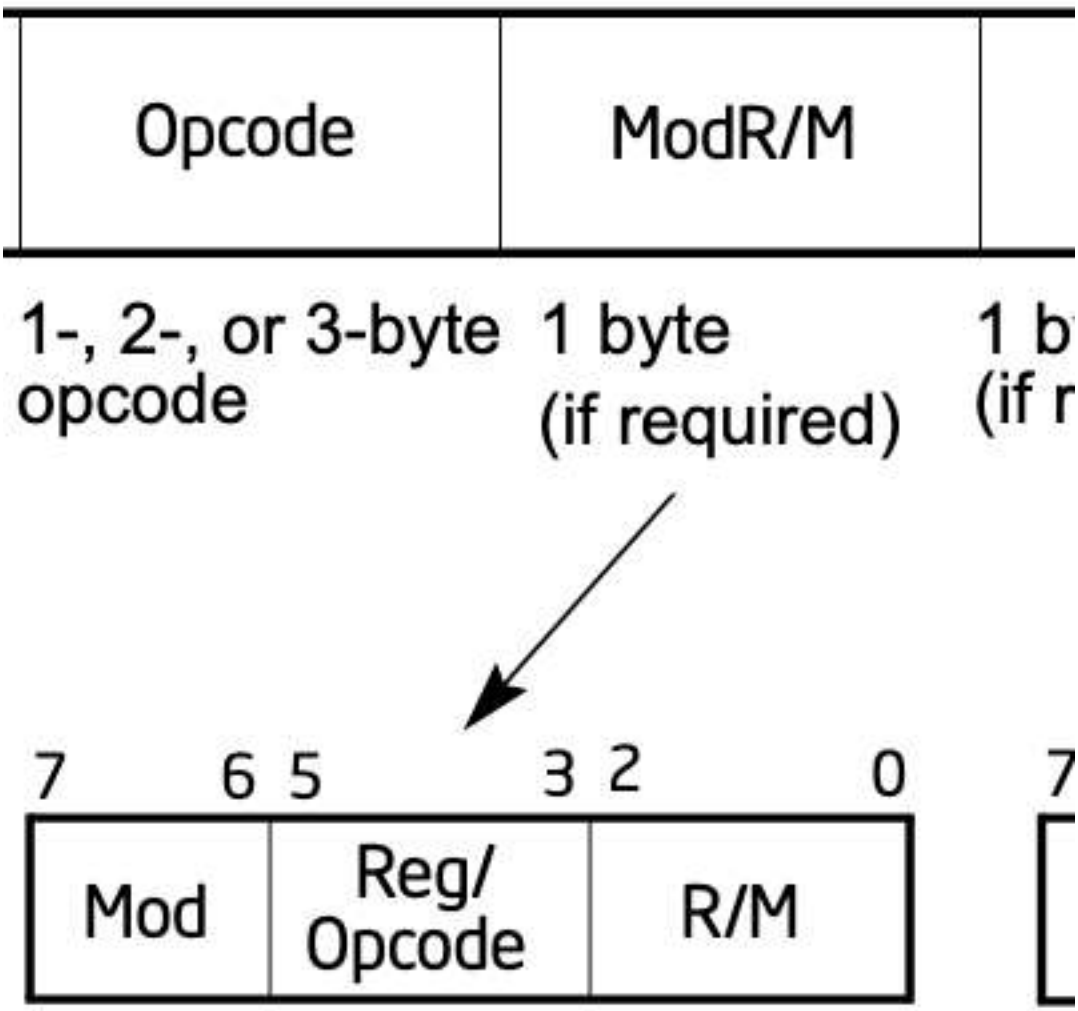
Se olharmos para a tabela, podemos ver que sempre que o valor do byte termina em 0 ou 8, eax é o primeiro operando:

EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Este modelo também se aplica a outros registros. Por exemplo, o ecx sempre termina com 1 ou 9, e o edx sempre termina com 2 ou A. Podemos ver que quando dividimos o operando por 8, esses valores são alinhados com o restante. Portanto, podemos usar o operador modular para recuperar nosso primeiro operando a partir do valor ModR/M:

```
modrm_value[Buffer[Localização]%8]
```

Para recuperar o segundo operando, podemos usar uma operação semelhante. Se olharmos para a estrutura ModR/M, podemos ver que o primeiro valor armazenado em 0, 1 e 2:



Por exemplo, quando convertido em binário, 0xD8 é representado como:

```
1101 1000
```

Para nosso primeiro operando, vemos três 000 bits associados à linha eax. Se movermos esses bits para a direita, obteremos os seguintes valores:

```
0001 1011
```

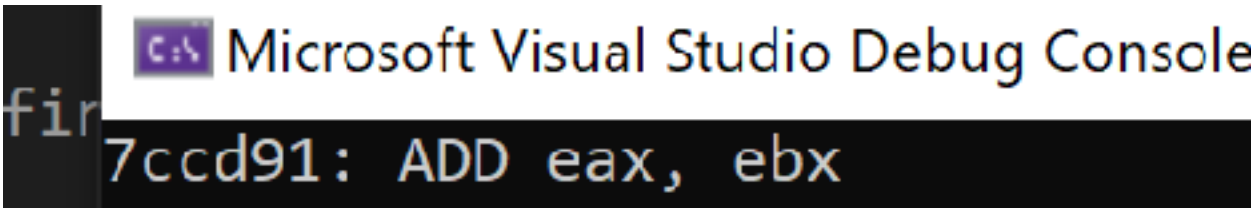
Se olharmos para a coluna no topo da tabela, 011 está associada à coluna ebx da mesma maneira que o primeiro operando. Então, quando mudamos o bits para recuperar o segundo operando por meio de um operador modular, podemos usar o mesmo método:

```
modrm_value[(Buffer [Localização]>>3)%8]
```

Com esses dois pontos, podemos realizar nossas funções:

```
if(buffer[Cocation]>=0xC0&buffer[Cocation]<=0xFF){printf("
%s, %s", modrm_value[Cocation]%8], modrm_value[(buffer[
Cocation]>>3)%8]);
Retornar 1;}
```

Executar este código imprimirá corretamente os operandos para a operação de adição:

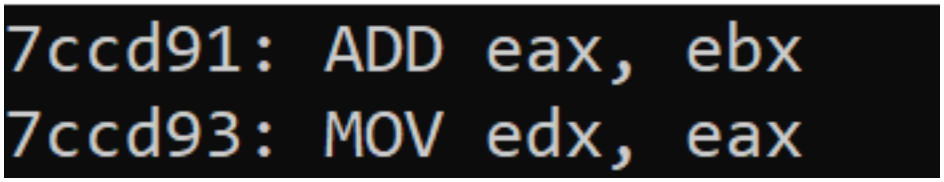


7.4.9 Outras notas

Agora que podemos desmontar as instruções add, podemos começar a processar outras instruções. Primeiro, vamos implementar o comando mov em 0x7cd93:

```
Caso 0x89:
printf ("MOV");
i++;
i+=decode_Opend (buffer, i);
descanso;
```

Ao executar isso, podemos verificar se nossa decodificação de operandos está funcionando corretamente, porque 0xC2 (operandos associados ao segundo movimento) é decodificado corretamente como edx, eax:



Entretanto, o próximo comando mov não foi decodificado corretamente. Embora seja o mesmo operando (0x89), ele tem um operando que nunca vimos antes, 0x85. Se analisarmos a tabela, descobriremos que isso está associado a [ebp]+deslocamento ou deslocamento. Se olharmos para a versão x64dbg, podemos ver que o deslocamento é -0x388. Sabemos que esse valor deve ser codificado em algum lugar na instrução. Como 0x89 85 foi considerado, o valor deve estar em 0x78fffff bytes.

No capítulo anterior, discutimos a ordem de bytes ou a ordem de bytes. Descobrimos que os bytes são armazenados em formatos pequenos. Portanto, precisamos reverter esses bytes:

```
FF FF FC 78
```

Este valor não corresponde a 0x388. Isso ocorre porque o valor é simbólico. Como este é um valor negativo, precisamos subtrair o valor máximo de um inteiro (0xFF FF FF) para obter o valor correto:

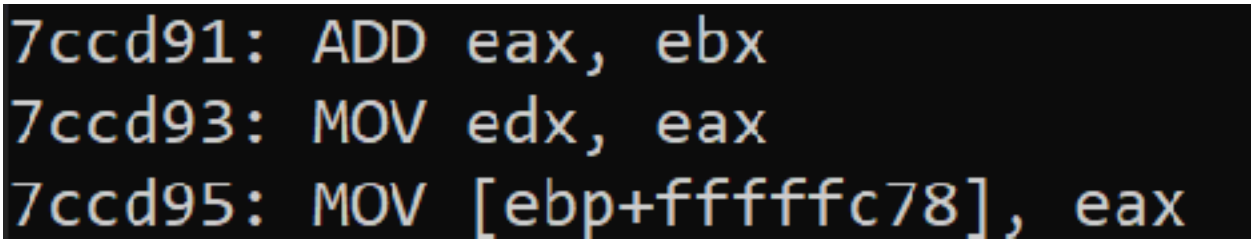
```
FF FF-FF FF FF
FC 78=387
```

Em seguida, precisamos adicionar 1 para explicar a mudança no símbolo e obter o valor correto -0x388.

Agora que entendemos como isso funciona, podemos adicioná-lo à operação de decodificação:

```
Esse if (buffer [Colocação]>=0x80&buffer [Colocação]<=0xBF){DWORD = buffer [Colocação]+1|(buffer [Colocação]+2]<<8) |
(buffer [Colocação]+3]<<16) (buffer [Colocação]<24;Priffer (Colocação][Colocação] + Colocação],[Colocação] (
Colocação],[Colocação], [Colocação][Colocação] %s][Colocação][Colocação], %s], %s] %s][Colocação] (Colocação], (
Colocação], (Colocação][Colocação] (Colocação], = % %s], %s], %s] %s] %s] %s] %s] <%s], (Colocação][Colocação][Colocação
], (Colocação], (Colocação] < <%s], %s], (Colocação], < < < %s], <=%s] (Colocação][Colocação], = = %s], (Colocação] = %s],


Retornar 5;}
```

Como vimos na primeira operação de decodificação, podemos usar bits de deslocamento para recuperar cada byte no deslocamento. Após incluir este conteúdo, a terceira ação agora pode ser decodificada corretamente:



7.4.10 Chamadas e saltos

Nos capítulos anteriores, descrevemos como usar as seguintes fórmulas para chamar ou operar jmp:

```
E8/E9 (NeW_Localização-Localização+5)
```

Podemos reverter esta operação e recuperar o endereço chamado do código de operação:

```
Caso 0xE8:
printf ("chamada");
i++;
loc=Buffer [i]|(Buffer [i+1]<8)|(Buffer [i+2]<<16)|(Buffer [i+3]<<24);

printf ("%x", loc+(i+(DWORD) me32.modbasidr)+4); i+=4;

descanso;
```

Adicionamos 4 em vez de 5 para ilustrar o fato de que o analisador excede 0xE8 bytes. No exemplo que escolhemos, também temos uma instrução curta de salto relativo if equal(je). Neste caso, podemos observar que o segundo byte do código de operação contém a quantidade a ser deslocada:

```
7cdA8 74 23 é 7cdcd 7cdDA8+23=
7cdcd
```

Também podemos adicionar essa lógica ao loop principal:

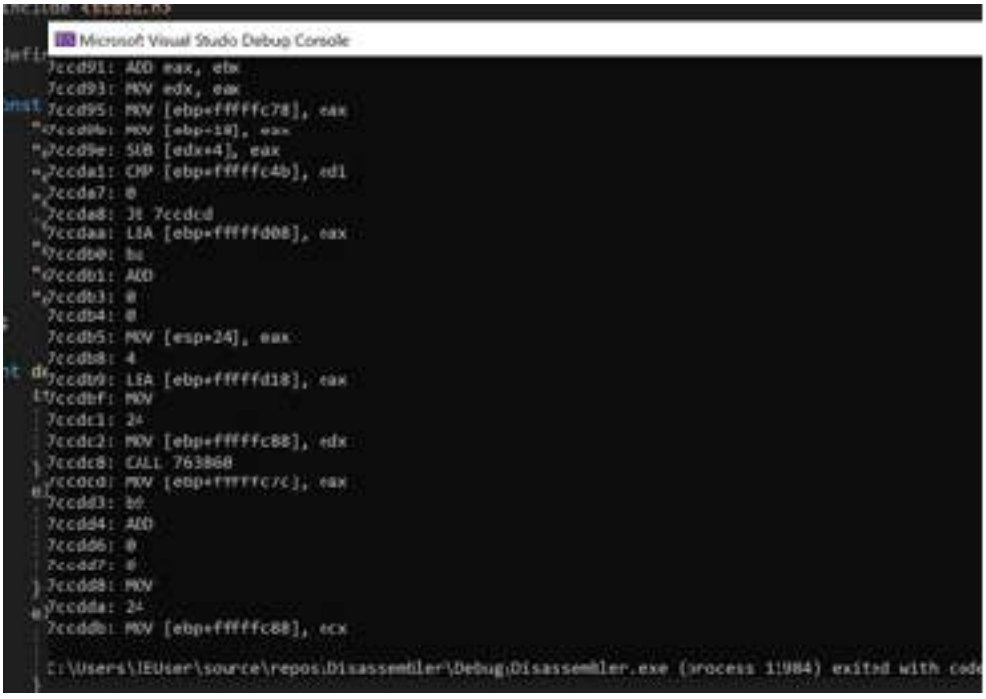
```
Caso 0x74:
printf("JE");
printf("%x", i+(DWORD) me32.modbasidr+2+ buffer [i+1]); i+=2;

descanso;
```

7.4.11 Resultado Final

Conforme declarado na isenção de responsabilidade, este não é um procedimento abrangente de descompilação. Na fonte mostrada no Apêndice A, os seguintes códigos de operação são implementados:

- Adicionar (0x01)
- MOV (0x89, 0x8B)
- Crianças (0x29)
- JE(0x74)
- Chamada (0xE8)
- CMP(0x80)
- Ler (0x8D)

Por meio dessas instruções, recuperamos os seguintes resultados:



7.5 Depurador

7.5.1 Objetivos

Nosso objetivo neste capítulo é atacar o Cubo Mágico 1,2.0,2.

7.5.2 Visão Geral

No capítulo anterior, usamos x64dbg para depurar e reverter o jogo. Após a fixação x64dbg Para esses jogos, podemos definir pontos de interrupção nos comandos do jogo. Quando o jogo executa esses comandos, nossos pontos de interrupção aparecem e a execução do programa é suspensa. Podemos então observar os valores de todos os registros e completar instruções individuais passo a passo.

Neste capítulo, exploraremos como usar a API Windows para criar depuradores para Windows. Vamos confirmar que o depurador está funcionando usando o Assault Cube como exemplo. No Capítulo 5.7, descobrimos que o comando mov do 0x0046366C só é executado quando o player está ligado. Depois que criamos o depurador, colocamos um ponto de interrupção nesta instrução e verificamos que ele só é atingido quando o atiramos.

7.5.3 API do depurador Windows

O Windows tem uma coleção de APIs que permite que um processo seja vinculado a outro processo e depurado outro processo. Estes são descritos em detalhes em vários artigos curtos no MSDN. Para nossos propósitos, estamos preocupados principalmente com as seguintes APIs:

- DebugAction Process, usado para anexar ao processo de destino
- WaitFordeGeVent, usado para aguardar eventos de depuração, conforme descrito no MSDN neste artigo
- ContinueDEGEVENT, usado para continuar a execução após o incidente de depuração ser acionado

Ao usar essas APIs, anexamos um processo e esperamos que ele acione um de vários eventos de depuração, como criar um thread ou encontrar uma exceção. Entretanto, ao depurar um alvo para o qual não temos código-fonte, isso nos limita a interromper eventos de criação de threads e processos.

Para acionar pontos de interrupção no endereço, precisamos usar instruções de interrupção . As instruções de interrupção são um conjunto especial de instruções de software usadas para chamar um manipulador de interrupção especial na CPU. Uma das instruções, int 3, acionará um ponto de interrupção durante a execução. Seu código de operação é 0xCC.

Podemos usar esse comportamento para definir pontos de interrupção em qualquer instrução. Antes de adicionar o depurador ao processo, usaremos o WriteProcessMemory para escrever 0 xCC na instrução que queremos interromper. Então, monitoraremos o evento de depuração como se estivesse normal. Quando recebemos um evento de ponto de interrupção, restauramos a instrução ao seu formato original e continuamos a executá-la. Ao fazer isso, podemos definir pontos de interrupção em qualquer instrução em um alvo que não é controlado pelo código-fonte.

O Apêndice A fornece o código-fonte completo do depurador discutido neste capítulo.

7.5.4 Escreva instruções Int 3

Para escrever instruções int 3 no destino, usaremos o método descrito nos capítulos anteriores. Primeiro, usaremos o Creatotolhelp32Snapshot para atravessar todos os processos no sistema e encontrar o processo de ataque ao conjunto de dados multidimensionais (ac_cliente.exe). Em seguida, abriremos o cabo do processo e usaremos esse cabo para escrever 0xCC (código de operação do int 3) nas instruções em 0x0046366C:

```
Mancha de frase process_snapshot=NULL;
Mancha de frase Process_Handlle=NULL;

DWORD pid
DWORD BYTE_WRTEN=0;

Byte instruction_break=0xcc;

processEntry32 pe 32={0};

pe32. dwSize = dwSize (PROCESSENTRY32);

process_snapshot=Creatotolhelp32snapshot (th32cs_snapshot, 0);
Process32First(process_snapshot, &pe32);

De {
Se (wcscmp (pe32. Szexeefile, L"ac_cliente.exe") ==0){pid=
pe32.th32 ProcessID;
```

```
process_handle=OpenProcess (PROCESS_ALL_ACCESS, verdade, pe32.
Th32Process);
WriteProcess_Handle, (void*) 0x0046366C, &instruction_break, 1, &
BYTE_Writeen);
}
}while (process_snapshot, &pe32);
```

Como usamos o identificador de processo (ou pid) que exige o cubo de ataque para a API DebugTiveProcess, também armazenamos o pid para uso futuro.

7.5.5 Ciclo do depurador principal

Em seguida, podemos usar o mesmo modelo discutido no MSDN para anexar alvos e lidar com eventos do depurador. O código fornecido no MSDN entra em um loop permanente que verifica o evento de depuração e continua a executá-lo quando ocorre um evento.

```
DEBUG_EVENT debugEvent={0};

DWORD Continuation = DBG_Continuation;

Processo de atividade de depuração (pid);

for(;;){
Continuestados = dbg_Continuestados;

Se (! debugEvent (&debugEvent, INFINITE))
Retornar a 0;

Interruptor (Debugevent.dwdebdeugeventcode){
Caso Excepcion_Debug_Event:
Interruptor (Debugevente.u.Exception Record.Exception Record.Exception
Code){
case EXCEPTION_BREAKPOINT:
Continuestados = dbg_Continuestados;
descanso;
Padrão padrão:
Continuestato=DBG_EXCEPTION_NOT_HANDLED; descanso;

}Descanso;
Padrão padrão:
```

```
Continuestato=DBG_EXCEPTION_NOT_HANDLED;
descanso;
}

ContinueeBogeVent (debugEvent. dwProcessId, debugEvent. dwProcessId,
ContinueeStat);
}

Caminho fechado (process_handle);
```

7.5.6 Processamento de pontos de interrupção

Com essa configuração estrutural, agora podemos começar a lidar com eventos do depurador. Primeiro, vamos verificar se o ponto de interrupção do int 3 funciona junto com a declaração print:

```
case EXCEPTION_BREAKPOINT:
printf ("ponto de ruptura acerto");

Continuestados = dbg_Continuestados
; descanso;
```

Certifique-se de que o Assault Cube esteja em execução e execute o depurador que construímos até agora. Ele deve imprimir imediatamente o ponto de interrupção. Se você atirar depois, ele imprimirá novamente um ponto de interrupção antes que o jogo caia. Isso mostra que nossa definição de ponto de interrupção foi bem-sucedida . Entretanto, atingir o alvo não foi o ideal. Para resolver esse problema, precisamos ajustar duas coisas:

1. O ponto de interrupção é acionado somente quando a instrução é executada, não quando o programa é executado pela primeira vez.
2. Restaure a instrução original após a execução do nosso ponto de interrupção.

Quando estamos vinculados a um processo pela primeira vez, uma exceção de ponto de interrupção é acionada . Como queremos lidar apenas com pontos de interrupção nas instruções, ignoraremos a primeira exceção:

```
bool first_break_has_occurred=false;
case EXCEPTION_BREAKPOINT:
if(first_break_has_occurred){
//Lidar apenas com eventos de ponto de interrupção após a primeira
exceção}
```

```
first_break_has_occurred=verdade;
```

Em seguida, podemos lidar com o colapso que ocorre quando o ponto de interrupção é acionado. Essa falha ocorreu porque substituímos o código de operação mov original (0x8b) por uma interrupção. Depois de executar nossa interrupção e processar o evento de depuração , o jogo tenta executar o próximo código de operação, mas isso é inválido. Para resolver esse problema, precisamos restaurar o comando mov após lidar com o evento de depuração.

Registros EIP (Ponteiro de Instruções Extendidas) são usados para rastrear instruções atualmente executadas. Toda vez que uma instrução é executada, ela é alterada para refletir o endereço da próxima instrução a ser executada. Quando executamos a instrução int 3, ela acrescenta 1. Para restaurar o comando mov, precisamos reduzi-lo primeiro.

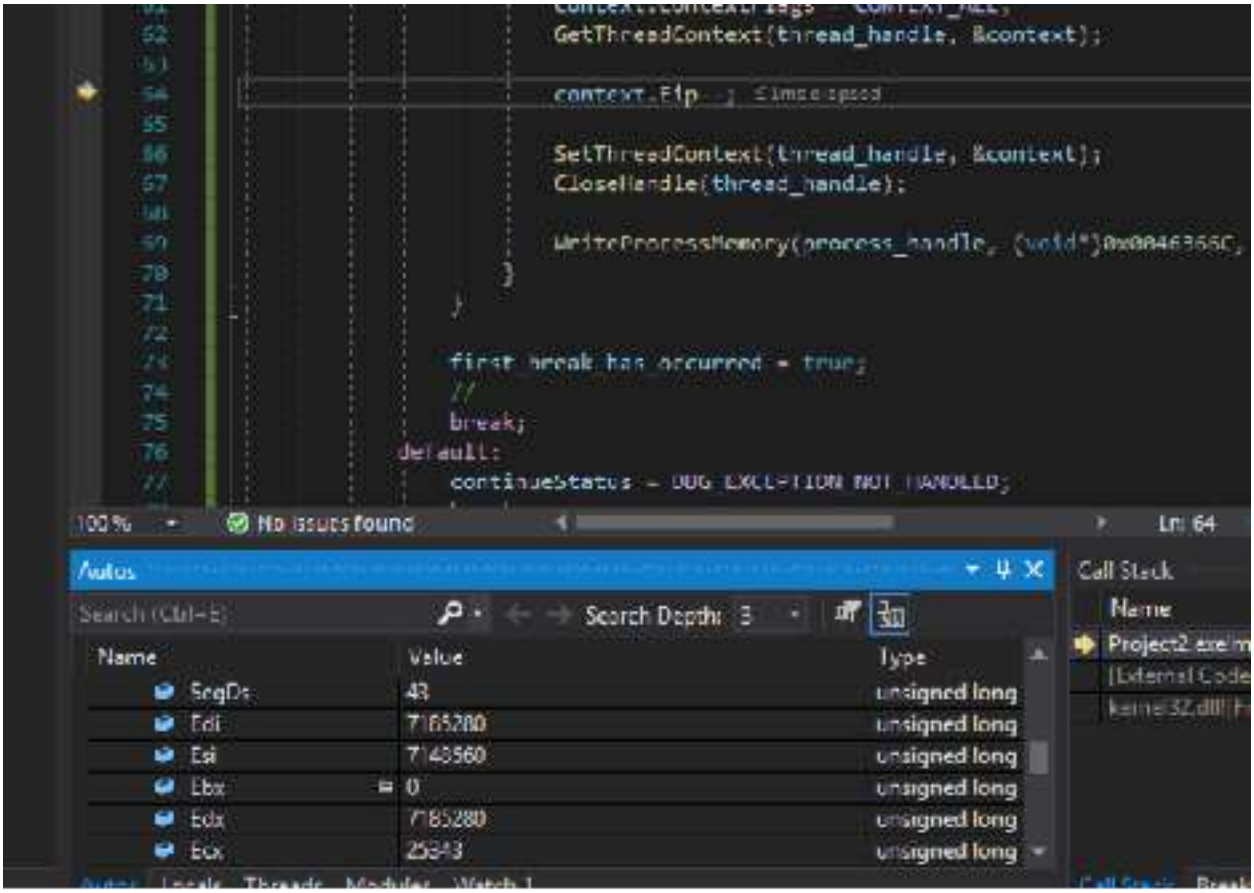
Podemos fazer isso abrindo a thread responsável por acionar o ponto de interrupção e recuperando o contexto (registro) da thread. Podemos então reduzir o registro EIP e definir o contexto do thread para um novo valor:

```
O cabo da frase é thread_handle=NULL;
Contexto contexto ={0};

thread_handle=OpenThread (Thread_All_acesso, verdade, debugevent.dwhread); Se
(thread_handle!=NULL){
Context.Contextflash=CONTEXT_ALL;
GettheadContext (thread_handle, &context);
Contexto. Eip-;


context (thread_handle, &context); CloseHandle (
thread_handle);
```

EIP agora apontará novamente para o endereço de comando mov original (0x0046366 C). Entretanto, o comando para esta posição ainda é int 3. Para resolver esse problema, podemos usar WriteProcessMemory para escrever o código de operação original de volta ao endereço:

```
WriteProcess_Handle, (void*) 0x0046366C, &instruction_normal, 1, &BYTES_
WRTESTEST);
```

Com essa mudança, quando nosso ponto de interrupção for acionado, o cubo de assalto não entrará mais em colapso. Além disso, podemos definir um ponto de interrupção no código de linha Context.ep-e verificar que quando nosso ponto de interrupção é acionado, podemos visualizar o conteúdo de todos os registros:



O mesmo método usado para modificar o EIP também pode ser usado para modificar outros registros.

7.6 Gravador de chamadas

7.6.1 Metas

Nosso objetivo neste capítulo é Wesnoth 1.14.9.

7.6.2 Visão geral

Ao reverter aplicativos complexos como videogames, uma das etapas mais difíceis é **criar um contexto dentro do aplicativo. Embora existam muitas técnicas de criação de contexto, uma abordagem** é **criar um depurador modificado que registra todas as instruções de chamada executadas pelo aplicativo. Você pode então executar ações no jogo, como clicar em botões, e observar todas as chamadas relacionadas. A chamada registrada pode então ser usada para criar contexto e começar a reverter o alvo.**

Nosso objetivo neste capítulo é **modificar o depurador que criamos no capítulo anterior para registrar todas as instruções de chamada emitidas pelo alvo. O código completo para este capítulo está disponível no Apêndice A.**

7.6.3 Posicionamento Módulo Principal

No capítulo anterior, escrevemos uma instrução de interrupção para uma posição no cubo de ataque. Nosso alvo neste capítulo é Wisner. Portanto, modificaremos o código responsável por localizar o processo pid para encontrar o processo Wesnoth e excluiremos o código responsável por escrever um único ponto de interrupção:

```
De {
Se (wcscmp (pe32.szoeefile, L"wesnoth.exe") == 0){pid=pe32.th32
processid;

process_handle=OpenProcess (PROCESS_ALL_ACCESS, verdade, pe32.
Th32Process);
}
}while(process_snapshot,&pe32);
```

Para esta ferramenta, registraremos apenas chamadas no módulo do jogo principal, não chamadas no DLL externo, como o user32.dll. Para determinar os endereços de início e término do módulo principal, primeiro usaremos a API EnumbrossModules para recuperar uma lista de todos os módulos carregados. Em seguida, usaremos a API GetModuleInformation Formation para recuperar o espaço de endereço do primeiro módulo, que sempre representa o módulo principal do jogo. Executaremos o seguinte código no primeiro evento de depuração que ocorre no destino (quando adicionarmos o depurador ao processo):

```
Módulo H Módulo [128]={0};
MODULEINFO module_info={0};

DWORD bytes_read=0;

if(!first_break_has_occurred){
Movimentos ENUMPOS (process_handle, modules, sizeof),
&bytes_read);
module_info (Proces_Handle, module_info [0], module_info (module_info
));
```

A API GetModuleInfo preencherá o Module_info.sizeoFage com o tamanho do módulo principal e preencherá o Module_info.lpbaseofll com o local do módulo principal. Com esse alcance, podemos começar a pesquisar instruções de chamada.

Como fizemos antes, usaremos a ReadProcessMemory para ler instruções no buffer. Embora queiramos ler toda a memória de todo o processo, esse método não funciona. Isso ocorre porque diferentes partes de memória de um processo têm diferentes proteções de memória. Se esta seção não for permitida para leitura, a chamada para o ReadProcessMemory falhará. Se tentarmos ler toda a memória do processo em uma única chamada, encontraremos uma parte falhada e toda a leitura falhará.
Para resolver esse problema, leremos a memória em segmentos. Essas partes são chamadas de páginas de memória, e o tamanho padrão da página de memória no Windows é de 4096 bytes. Portanto, criaremos um loop para ler 4096 bytes de instruções de uma só vez. Usaremos o parâmetro bytes_read para determinar quantos bytes a página realmente lê:

```
#define Read_página_size 4096

Instrução de caracteres sem símbolos [read_página_tamanho]={0};

for(DWORD i=0;i<move_info.sizeofige;i+=read_página_tamanho){readProcess_
Handle((LPVOID)((DWORD) move_info.lpbase+i), &infos,read_size_size_size_size_
size_size_size_size_size_ i);
```

```
for(DWORD c=0;c<bytes_read;c++){}


}
```

7.6.4 Chamada de localização

Em seguida, encontraremos instruções de chamada em cada página da memória . Sabemos que o código de operação do comando de chamada é 0xe8. Ao iterar cada instrução, verificaremos se ela é 0xe8:

```
Byte Instruction_Chamada=0xe8;
...
for(DWORD c=0; c< Byte_Read; c++) {if(
instructions[c]==Instructions_Call){}


}
```

Entretanto, nem todos os 0xe8 representam instruções de chamada. Por exemplo, add eax, o código de operação da instrução ebp é 0x01 e8. Precisamos garantir que esses 0xe8 aleatórios não sejam reconhecidos como chamadas. A maneira mais simples é ler 4 bytes após a chamada.

Como sabemos no Capítulo 2.6, esses 4 bytes codificam a localização da chamada. Ao pesquisar esta localização, podemos verificar se a localização calculada desses bytes é válida. Caso contrário, podemos presumir que 0xe8 não é uma chamada e usar a instrução continue para evitar esta verificação:

```
Deslocamento DWORD = 0;
DWORD Call_Localização=0;
DWORD Call_Localiz_BYTES_Read=0;
...
Se(instrução [c]==Instruction_chamada){
Oferta =(DWORD)Module_info.lpbasseofdll+i+c;
ReadProcess_Handle, (LPVOID) (offset+1), &Call_Localiza,4.&Call_Localiza_
Localiza_Localiza_Localiza_Localiza_Localiza_Localiza_);

Call_Localização+=Oferta+5;
Se (call_localização <(DWORD)move_info.lpbaseofdl||call_localização>(
DWORD)move_info.lpbaseofdll+move_info.sizeofige)
```

```
Continuar;
```

Por fim, gravaremos um comando de interrupção (0xcc) para esse local. Além do WriteProcessMemory, também usaremos a API FlushinStruckwork para garantir que nossas mudanças sejam realizadas imediatamente em direção ao alvo:

```
Instrução de byte _break=0xcc;
...
WriteProcess_Handle, (void*) Oferta, &instruction_break, 1, &BYTES_
WRTESTEST);
FlushinstructionCase (Proces_Handle, (LPVOID) offset, 1);
```

Escrever milhares de instruções de interrupção em um processo pode causar a falha do programa. Para evitar isso, escreveremos apenas 2.000 pontos de interrupção:

```
int breakpoints_set=0;
...
Se (breakpoints_set<2000){
WriteProcessMemory...
breakpoints_set++;
}
```

7.6.5 Processamento de pontos de interrupção

Agora que escrevemos pontos de interrupção para todas as chamadas, precisamos lidar com eventos de pontos de interrupção. Começaremos com a mesma abordagem usada no capítulo anterior:

```
Caso contrário{
thread_handle=OpenThread (Thread_All_acesso, verdade,
debugevent.dwhread);
Se (thread_handle!=NULL){
Context.Contextflash=CONTEXT_ALL; GettheadContext (
thread_handle, &context); Contexto. Eip-;



context (thread_handle, &context); CloseHandle (thread_
handle);
```

```
WriteProcessMemory (process_handle, (void*)context.Eip,
&Instruction_Chamado, 1, &BYTE_WRTEN);
FlushinstructionCase (process_handle, (LPVOID)context.Eip, 1);}

}
```

Como vimos anteriormente, esse código reduzirá o EIP e restaurará as instruções de chamada originais. A execução será então restaurada na chamada e o programa continuará a ser executado normalmente. A desvantagem desse método é que cada ponto de interrupção é atingido apenas uma vez. Para nossos gravadores de chamadas, queremos registrar cada chamada executada. Para alcançar esse comportamento, usaremos um modo de passo único.

Etapa única é um tipo especial de evento de depuração que executa uma única instrução antes que uma exceção seja acionada novamente. Para habilitar o modo passo único, modificamos o EFlags do thread atual da seguinte forma:

```
Contexto. Eip-;
Contexto. EFlags|=0x100;
```

Em seguida, precisamos lidar com eventos de uma única etapa. Apresentaremos outro caso para isso:

```
Estudo de caso Excepcion_Single_Passo:
```

Quando recebemos uma exceção aqui, isso significa que a chamada foi executada. Em última análise, nosso objetivo neste incidente é restaurar o comando break. Podemos fazer isso através do WriteProcessMemory da mesma maneira que recuperamos as instruções de chamada:

```
thread_handle=OpenThread (Thread_All_acesso, verdade, debugevent.dwhread); Se
(thread_handle!=NULL){
context.ContextFlags=CONTEXT_ALL;
GettheadContext (thread_handle, &context);
CloseHandle (thread_handle);

WriteProcess_Handle, (void*)last_call_location,
&instruction_break, 1, &BYTES_WRTESTEN);
FlushinstructionCase (process_handle, (LPVOID)Last_Call_Localizone, 1);}
```

Com esse código, nossos pontos de interrupção serão restaurados após o acionamento.

7.6.6 Adicionar registros

Por fim, adicionaremos registros a este código para que possamos ver o ponto de interrupção do acionamento. Em nosso evento de depuração, armazenaremos a localização atual do EIP:

```
DWORD Last_Call_Localização=0;
...
Last_Call_Localização=context.Eip;
```

Em seguida, no evento de passo único, adicionaremos um código de registro. Neste ponto , sabemos que executamos a chamada e estamos no local da chamada. Agora, podemos usar a seguinte declaração print para imprimir o endereço e a localização da chamada:

```
printf("0x%08x: chamada 0x%08x\n", last_call_localiza,context.Eip
); Last_Call_Localização=0;
```

Neste capítulo, registramos apenas as chamadas que ocorrem. No entanto, esse código também pode ser modificado para vincular a instrução ret. Isso permitirá que você crie um gráfico mostrando todas as chamadas feitas pelo processo e quais chamadas chamaram outras chamadas.

Apêndice A

Script de configuração VM do laboratório A1

Veja Capítulo 1.4.

```
Set-Windows Sextorprograms-Inableshodfiles Foldersdrives-
Inableshodfiles-Inableshodfiles-Inableshodfiles
Ative - Desktop remoto

Mecanismo de trapaça
cinst x64dbg. Portátil
```

A.2 Wisno Externo Hacker de ouro

Veja Capítulo 3.2.

Wesnoth 1.14.9 Hacker externo, modificando o ouro do jogador.

Este código criará um aplicativo de console que definirá o ouro do jogador no Wesnoth 1.14.9 para 555 no tempo de execução. Ele usa ReadProcessMemory e WriteProcessMemory para fazer isso. O endereço 0x017EED18 representa o ponteiro básico do jogador no Wesnoth.

Este programa deve ser executado como administrador.

```
//FindWindows, GetWindows ReadProcess, OpenProcess, ReadProcess e WriteProcess
estão todos incluídos no windows. h.
#include<Windows.h>

int main(int argc, char**argv){
```

```
/*
Para usar o ReadProcessMemory e o WriteProcessMemory, precisamos de um cabo para o
processo Wesnoth.

Para obter esse cabo, precisamos de um id de processo. A maneira mais rápida de obter
um id de processo específico
O processo é feito usando o GETWindows ReadProcess.

O GetWindows ReadProcess requer um cabo de janela (diferentemente do cabo de
processo). Para obter isso
Para alças de janela, usamos FindWindows.
*/

//Encontre nossa janela Wesnoth. Isso pode variar dependendo das configurações do
seu idioma.
HWND Wesnoth_Windows=FindWindows(NULL, L"Wesnoth Battle-
1.14.9");

//Obtenha o id do processo do processo Wesnoth. O GetWindows ReadProcess não retorna
o id do processo, mas
//Em vez disso, preencha-o com o valor da variável fornecida, então &.
DWORD Proces_ID=0;
GetWindows ReadProcess (wesnoth_Windows, &process_id);

//Abra nosso processo Wesnoth. PROCESS_ALL_ACCESS significa que podemos ler e
escrever processos. Entretanto,
//Isso também significa que o programa deve ser executado como administrador para
funcionar.
Manguel de frase Wesnoth_Pross=OpenPross(Pross_All_Access, verdade,
process_id);

//Leia o valor em 0x017EED18 e coloque-o na variável gold_value.

DWORD gold_value=0;
DWORD bytes_read=0;
ReadProces_Process, (void*) 0x017EED18, &gold_value, 4,
&bytes_read);

//Adicione 0xA90 ao valor lido na etapa anterior e leia o valor no novo endereço.
Estes
//offsets estão incluídos em https://gamihacking.academy/lesson/13
gold_value+=0xA90;
ReadProcess_Process, (void*) gold_value, &gold_value, 4, &bytes_read);
```

```
//Adicione gold_value a 4, e esse valor apontará para o endereço dourado atual do
jogador.
//Escreva o valor new_gold_value (555) para este endereço
gold_value+=4;
DWORD new_gold_value=555;
DWORD BYTE_WRTEN=0;
WriteProcess_Process_Process_Process_Process_Process_Process_Process_
Process_Process_Process_Process_Process_Process_, (void*)

Retorne a 0;}
```

A.3 Wesnoth Interno Hacker de ouro

Veja Capítulo 3.3.

Hacker de memória do Wesnoth 1.14.9, modificando o ouro do jogador.

Este é um exemplo de um DLL que precisa ser injetado com Wesnoth. Uma vez injetado, ele cria uma thread no jogo. Este thread espera que o jogador pressione a tecla "M" e, em seguida , usa uma série de ponteiros para definir diretamente o valor dourado do jogador para 999.

Isso deve ser injetado no processo Wesnoth para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
//Creathead e GetAsyncKeyState são definidos em windows.h.
#include<Windows.h>

//Nosso thread de injeção. Como queremos monitorar as teclas do usuário, // usamos o
loop while para garantir que o thread nunca saia. Dentro do thread, nós

//Verifique se a tecla "M" está pressionada e segura. Se for o caso, acessamos
diretamente a memória do jogo
//Usando ponteiros. Usamos esses ponteiros para definir o valor ouro do jogador.

void injectid_THREAD(){
```

```
      while(true){
            if(GetaseNCKETATE('M'){
                    DWORD*player_base=(DWORD*)0x017EED18;
                  DWORD*tame_base=(DWORD*)(*player_base+0xA90);
                  DWORD*Good=(DWORD*)(*game_base+4);
                  *Ouro = 999;
            }

            //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
             //Isso permite que o processador agende outras tarefas.
            Sono(1);
      }
}

//Durante a injeção, o processo pai busca o DLLL DLL, semelhante à função main em
arquivos executáveis regulares.
//Existem vários eventos que podem acontecer, o mais importante para nós é DLL_
PROCESS_ATTACH. Quando
//DLL é totalmente carregado na memória do processo.
//
//Uma vez carregado, criamos um thread. Enquanto o processo permanecer aberto, o
thread será executado em segundo plano do jogo.
//O código que o thread executará é mostrado acima.
BOOL WINAPI DllMain(Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      Se(fdwReason==DL_Process_Attach){
            Reatethread(NULL, 0, (lpthead_star_routine) injected_thread, NULL,
0, NULL);
      }

      Retornar a
verdade;}
```

Código A.4 Wesnoth cavernas, etc.

Veja Capítulo 3.4.

Um DLL redireciona a função de descrição do terreno em Wesnoth 1.14.9 para uma função personalizada, definindo o ouro do jogador para 888.

Esta função personalizada então recria a função de descrição do terreno e devolve a execução ao programa.

Isso é feito usando o código caverna. Ao injetar, a DLL modifica a função que exibe a descrição do terreno e altera o código para pular para a função de caverna de código definida na DLL. A função Code Cave então salva o registro, define o gold para 888 e restaura a instrução original modificada antes de retornar ao código de chamada original.

Isso deve ser injetado no processo Wesnoth para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>

DWORD*player_base;
DWORD*gami_base;
DWORD* Ouro;
DWORD Ret_Adress=0xCCAF90;

//O programa executa a caverna de código para o qual saltará. A propriedade
declspec naked diz ao compilador para não adicionar nenhuma função
//Cabeças ao redor do código de montagem
__declspec(naked)void codecave(){
	//Os blocos Asm permitem que você escreva montagens puras
	//Neste caso, usamos isso para salvar todos os registros
	_asm{
		Pushad
	}

	//Defina o ouro do jogador de acordo com o mesmo método discutido em https
://gamehacking.Adamy/lesson/16
	player_base=(DWORD*)0x017EED18;
	game_base=(DWORD*)(*player_base+0xA90);
	Good=(DWORD*)(*tame_base+4);
	*Ouro = 888;

	//Restaurar o registro e recriar as instruções originais que
sobreponhamos
	//Depois disso, salte de volta à instrução que reescrevemos _asm{

		Rebote
```

```
            mov eax, dword ptr ds: [ecx]
            lea esi, dword ptr ds: [esi]
            jmp Ret_Adress
      }
}

//Quando nosso DLL é anexado, a proteção da memória é removida no código que queremos
escrever
//Em seguida, defina o primeiro código de operação para E9 ou salte
//Usando a fórmula para calcular a posição: New_Locação-
Localização original +5
//Por fim, como a instrução original tem um total de 6 bytes, o NOP emite os últimos
bytes restantes.
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      DWORD old_protect;
      unsigned char*Hook_Localização=(unsigned char*)0x00CCAF8A;

      Se(fdwReason==DL_Process_Attach){
            Projeto Virtualítico ((void*)Hook_Locasionismo, 6. Página _
Locasionismo, &ol_Projeto);
            *Hook_Localização=0xE9;
            *(DWORD*)(hook_localização+1)=(DWORD)&
codecavidade-((DWORD)hook_localização+5);
            *(hook_localização+5)=0x90;}


      Retornar a
verdade;}
```

A. 5 Visnos Starhak

Veja Capítulo 4.1.

Wesnoth 1.14.9 Stattack exibe a moeda de ouro do segundo jogador sempre que a caixa de descrição do terreno é exibida.

Isso é feito usando o código caverna. Ao injetar, a DLL modifica a função que exibe a descrição do terreno e altera o código para pular para a função de caverna de código definida na DLL. A função code cave então salva o registro, obtém a moeda de ouro do segundo jogador e escreve o valor no buffer do jogo para exibir o texto de descrição do terreno. Ele então salta de volta ao método de descrição do terreno e exibe a descrição original com uma cor dourada.
Isso deve ser injetado no processo Wesnoth para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>
#Incluindo <stdio.h>

DWORD*player_base;
DWORD*gami_base;
DWORD* Ouro;

//Endereço original da chamada do jogo
DWORD or_Call_adress=0x5E9630;

DWORD Ret_Adress=0x5ED12E;

//Buffer para salvar o valor de ouro do segundo jogador
char Gold_Bytte_Array[4]={0};

//O programa executa a caverna de código para o qual saltará. A propriedade
declspec naked diz ao compilador para não adicionar nenhuma função
//Cabeças ao redor do código de montagem
__declspec(naked)void codecave(){
      //Os blocos Asm permitem que você escreva montagens puras
      //Neste caso, usamos isso para salvar todos os registros
      _asm{
            Pushad
      }

      //Obtenha o valor dourado do segundo jogador de acordo com o ponteiro
      básico player_base=(DWORD*)0x017EED18;
      game_base=(DWORD*) (*player_base+0xA90);
      gold=(DWORD*) (*gami_base+0x274);

      //Converter o valor gold para seu ASCII, representado
      SRINTF_S (gold_byte_array,4, "%d", *gold);

    //Restaurar registros danificados pelo sprintf e salvá-los novamente
```

```
	//Em seguida, carregue o buffer do edx e coloque cada byte do ouro do segundo
jogador
	//Valor para buffer
	_asm{
		Rebote
		Pushad
		mov eax, dword ptr ds: [edx]
		mov bl, gold_byte_array[0]
		mov Byte ptr ds: [eax], bl
		mov bl, gold_byte_array[1]
		mov Byte ptr ds: [eax+1], bl
		mov bl, Gold_Bytte_Array[2]
		mov Byte ptr ds: [eax+2], bl
	}

	//Restaurar o registro e recriar as instruções originais que sobreponhamos

	//Depois disso, salte de volta à instrução que reescrevemos _asm{

		Rebote
		Ligue para ori_Call_Adress
		jmp Ret_Adress
	}
}

//Quando nosso DLL é anexado, a proteção da memória é removida no código que queremos
escrever
//Em seguida, defina o primeiro código de operação para E9 ou salte
//Calcule a posição usando a seguinte fórmula: New_Localização-Originação+5

BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

	DWORD old_protect;
	unsigned char*Hook_Localização=(unsigned char*)0x5ED129;

	Se(fdwReason==DL_Process_Attach){
		Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
		*Hook_Localização=0xE9;
		*(DWORD*)(hook_localização+1)=(DWORD)&
codecavidade-((DWORD)hook_localização+5);
	}

	Retornar a
verdade;}
```

A.6 Mapa Wesnoth Huck

Veja Capítulo 4.2.

O hacker de mapas do Wesnoth 1.14.9 revela o mapa inteiro eliminando a névoa de guerra no jogo.

Isso é feito modificando o código do jogo responsável por redefinir todas as cartas para um estado oculto no início do turno do jogador. Modifique este código para configurar todos os blocos visíveis (-1, ou 0xffffffffff em Wesnoth). Para acomodar o espaço das instruções anteriores, isso é feito usando ou dword ptr ds: [esi], instruções 0xFFFFFFFF (código de operação 0x830EFF) e vários NOPs (0x90).

Isso deve ser injetado no processo Wesnoth para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>

//Novo código de operação para gravar o código do jogo
unsigned char new_bytes[8]={0x90,0x90,0x90,0x83,0x0E, 0xFF,0x90,0x90};


//Quando nosso DLL for adicionado, primeiro desproteja-o. Responsável por
redefinir a memória dos azulejos do jogo.
//Em seguida, escreva nosso novo código de operação neste local de memória
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      DWORD old_protect;
      unsigned char*Hook_Locação=(unsigned char*)0x6CD519;

      Se(fdwReason==DL_Process_Attach){
            Projeto Virtualítico ((void*)Hook_Locasionismo, 8. Págão_
Exploração, &ol_Projeto);
            for(int i=0; i<sizeof(new_bits); i++){*(hook_
                  localização+i)=new_bits[i];
            }
            }
```

```
        Retornar a verdade;
}
```

A.7 Longsun Macro Robô

Veja Capítulo 4.3.

Um hacker para a versão Wyrmsun 5.0.1 criará automaticamente uma unidade de trabalho a partir da estrutura atualmente selecionada quando as moedas de ouro do jogador excedem 3.000.

Ele faz isso preenchendo o buffer da célula atual com dados de trabalho e então chamando a função create unit no jogo.

Depois de injetar esse hacker, entre no jogo e recrute um trabalhador. Em seguida, escolha uma estrutura ao coletar ouro. Você notará que os trabalhadores fazem fila automaticamente. Devido à maneira como Wyrmsun lida com o recrutamento, unidades podem ser criadas de qualquer unidade selecionada, incluindo outras.

Isso deve ser injetado no processo Wyrmsun para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>

O cabo de frase wyrmsun_base;

Cardinal DWORD*;
DWORD*unitbase;
DWORD RET_Unit_RET_Adress; DWORD
Recruit_Unit_Call_Adress;
Caracteres sem símbolo unitdata[0x
110]; Inicialização booleana = falsa;

DWORD GameOP_Ret_Adress; DWORD
Gameloop_Call_Adress; DWORD*gold
_base, *gold;
```

```
//Código da unidade de recrutas Função da unidade de recrutas no jogo Cave Hook
//Sua principal função é copiar um buffer de dados válido para a unidade de trabalho
//Em vez de ter que invertir a estrutura
__declspec(naked)void recruit_unit_codecave(){
_asm{
Pushad
Base mov, ecx
}

unit base=(DWORD*)(*base);
memcpy (unitdata, unitbase, 0x110);
init=verdade;

_asm{
Rebote
Impulsionar ecx
MiECX, esi
Ligue para recruit_Unit_Call_Adress
jmp Recruit_Unit_Ret_Adres
}
}

//No loop principal do jogo, nosso código caverna verifica o ouro do jogador atual
// Se exceder 3.000 e temos um buffer de trabalhadores válido, chamando a função
recruit unit
//Contém dados de trabalhadores.
__declspec(naked)void gameloop_codecave(){
_asm{
Pushad
}

gold_base=(DWORD*)((DWORD)wyrmsun_base+0x0061A504); gold=(
DWORD*) (*gold_base+0x78);
Ouro =(DWORD*) (*ouro + 4);
Ouro =(DWORD*) (*ouro + 8);
Ouro =(DWORD*) (*ouro + 4);
Ouro =(DWORD*) (*ouro);
gold=(DWORD*) (*gold+0x14);

if(init&&*gold>3000){memcpy(unitbase,
unitdata, 0x110); _asm{

mov ecx, empurrão básico
ecx
```

```
Ligue para recruit_Unit_Call_Adress
}
}


_asm{
Rebote
Chamar Gameloop_Call_Adress
jmp Gameloop_Ret_Adress
}
}

//Quando nosso DLL é anexado, a proteção da memória é removida no código que queremos
escrever
//Em seguida, defina o primeiro código de operação para E9 ou salte
//Usando a fórmula para calcular a posição: New_Locação-
Localização original +5
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      DWORD old_protect;

      Se(fdwReason==DL_Process_Attach){
            //Devido à carga dinâmica do código Wyrmsun, precisamos calcular o
deslocamento com base no endereço do módulo principal.
            wyrmsun_base=GetModuleHandle (L"wyrmsun.exe");

            unsigned char*Hook_Locação=(unsigned char*)
((DWORD)wyrmsun_base+0x223471);
            recruit_unit_ret_address=(DWORD)hook_location+8; Recrut_
            Unit_Call_Adress=(DWORD)wyrmsun_base+0x2CF7;

            Projeto Virtualítico ((void*)Hook_Locasionismo, 8. Págão_Exploração,
&ol_Projeto);
            *Hook_Localização=0xE9;
            *(DWORD*)(hook_localização+1)=(DWORD)&recruit_unit_codecave-((
DWORD)hook_localização+5);
            *(hook_localização+5)=0x90;
            *(hook_localização+6)=0x90;
            *(hook_localização+7)=0x90;

            Hook_Localização=(unsigned char*)((DWORD)wyrmsun_base+0x385D34);
            gameloop_ret_address=(DWORD)hook_localização+5;
               Gameloop_Call_Adress=(DWORD)wyrmsun_base+0xDBCA;
```

```
            Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
            *Hook_Localização=0xE9;
            *(DWORD*)(hook_localização+1)=(DWORD)&gamioop_codecave-((
DWORD)hook_localização+5);
      }

      Retornar a
verdade;}
```

A. 8 Terror Urbano

Hacker de muros de memória

Ver Capítulo 5.2.

O wallhack, do City Terror 4.3.4, revelou entidades na parede ao desativar testes de profundidade.

Isso é feito modificando a bandeira de renderização de cada entidade, que é responsável por determinar como a entidade deve ser renderizada. Ao definir esse valor para o valor intra-jogo (0xD) que desativou o teste profundo, as entidades serão desenhadas, independentemente de serem visíveis ou não. O código vinculado é uma instrução mov que ocorre depois que o ebx carregar uma estrutura de entidade válida. Isso deve ser injetado no processo de terror urbano para desempenhar um papel. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>

DWORD Ret_Adress=0x0052D303;

//O programa executa a caverna de código para o qual saltará. A propriedade
declspec naked diz ao compilador para não adicionar nenhuma função
//Cabeças ao redor do código de montagem
__declspec(naked)void codecave(){
      //Os blocos Asm permitem que você escreva montagens puras
      //Neste caso, usamos isso para salvar todos os registros
```

```
	//Em seguida, defina o valor de renderização da entidade [ebx+4] para
desativar o teste de profundidade (0xD)
	//Em seguida, restauramos o registro, recriamos a instrução original e
saltamos de volta para o código do programa
	_asm{
		Pushad
		mov dword ptr ds: [ebx+4], 0xD
		Rebote
		mov dword ptr ds: [0x102AE98], ebx

		jmp Ret_Adress
	}
}

//Quando nosso DLL é anexado, a proteção da memória é removida no código que queremos
escrever
//Em seguida, defina o primeiro código de operação para E9 ou salte
//Usando a fórmula para calcular a posição: New_Locação-
Localização original +5
//Por fim, como a instrução original tem um total de 6 bytes, o NOP emite os últimos
bytes restantes.
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

	DWORD old_protect;
	unsigned char*hook_localização=(unsigned char*) 0x0052D2FD;

	Se(fdwReason==DL_Process_Attach){
		Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
		*Hook_Localização=0xE9;
		*(DWORD*)(hook_localização+1)=(DWORD)&
codecavidade-((DWORD)hook_localização+5);
		*(hook_localização+5)=0x90;}


	Retornar a
verdade;}
```

A9 Terror Urbano OpenGL Wallhack

Ver Capítulo 5.3.

O wallhack do City Horror 4.3.4 exibe entidades através da parede ao ganhar a função OpenGL do jogo glDrawElements e desativar o teste de profundidade do OpenGL.

Isso é feito localizando a função glDrawElements na biblioteca OpenGL e criando uma caverna de código no início da função. Em uma caverna de código, examinamos o número de vértices associados a um elemento. Se exceder 500, chamamos o glDepthRange para limpar o plano de corte profundo e chamamos o glDepthFunc para desativar o teste profundo. Caso contrário, chamamos essas mesmas funções para reativar o plano de recorte de profundidade e reativar o teste de profundidade.
Este DLL deve ser injetado no processo de terror urbano para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>

HMODULE Openglhandele=NULL;

//Preenchemos dinamicamente os ponteiros de função das duas funções OpenGL

//Depois de injetar nossa DLL
void(__STDCal*GLDEPTHUNC)(unsigned int)=NULL;
void(__STDCal*GLDEPTHRGE)(double, double)=nul;

unsigned char*hook_localização;

DWORD Ret_Adress=0;
DWORD old_protect;
Contagem DWORD = 0;

//Cave de código executada antes de chamar glDrawElements
__declspec(naked)void codecave(){
    //Primeiro, recuperamos os parâmetros de contato da chamada original.
```

```
	//Em seguida, buscamos o valor do parâmetro de contato, que especifica a
quantidade
	//Índice a ser apresentado
	_asm{
		Pushad
		mov eax, dword ptr ds: [esp+0x10]
		Contagem de metros, eax
		Rebote
		Pushad
	}

	//Se a contagem exceder 500, limparemos o plano de corte de profundidade e //
	definimos a função de profundidade para GL_ALWAYS
	Se (contar > 500){
		(*GLDEPTHRGE)(0,00,0);
		(*glDepthFunc)(0x207);
	}
	Caso contrário{
		//Caso contrário, restaure o plano de corte profundo ao valor padrão do
jogo e, em seguida,
		//Defina a função de profundidade para GL_LEQUAL
		(*GLDEPTHRGE)(0,0,1,0);
		(*glDepthFunc)(0x203);
	}

	//Por fim, restaure o comando original e salte de volta
	_asm{
		Rebote
		mov esi, dword ptr ds: [esi+0xA18]
		jmp Ret_Adress
	}
}

//Responsável por criar threads de injeção de gancho
void injectid_THREAD(){
	while(true){
		//Como o OpenGL é carregado dinamicamente no processo, nossos threads
precisam esperar
		//Até ver que o módulo OpenGL foi carregado. Se(Openglhandel=
		=NULL){
			Opengll e=GetModuleHandle (L"opengl32.dll");
		}

		Uma vez carregado, primeiro encontramos a localização das duas funções
de profundidade que estamos usando
```

```
            //Caverna de código acima
   if(Open GlDHANDLE!=NULE & GlDHANDLE ==NULE){GlDHANDLE =(void(__StDHANDL*)(
                                    unsigned)
int) GetProcadodres (openGLHandle, "glDepthFunc");
                    Gldepathrange=(void(__stdcal*) (double,
double) Gldepathrange (Opengldepathrange, "Gldepathrange");

                    //Então encontramos a posição do glDrawElements e o deslocamos
para um comando fácil de ganhar
                    hook_location=(Sem símbolo
char*) GetProcaderse (Openglândle, "glDrawElements");
                    Hook_Localização+=0x16;

                    //Para ganchos, removemos a proteção da memória no código que
queremos escrever
         //Em seguida, defina o primeiro código de operação para E9 ou salte
                    //Usando a fórmula para calcular a posição: New_Localização-
Originação+5
                    //Por fim, como a primeira instrução original tem um total de 6
bytes, o NOP emite os últimos bytes restantes
                    Projeto Virtual* (void*)Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
                    *Hook_Localização=0xE9;
                    *(DWORD*)(hook_localização+1)=(DWORD)&codeave-
((DWORD)hook_localização+5);
                    *(hook_localização+5)=0x90;

                    //Como o OpenGL é carregado dinamicamente, precisamos calcular
dinamicamente o endereço de retorno
                   Ret_Adress=(DWORD) (hook_localização+0x6);
             }

             //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
              //Isso permite que o processador agende outras tarefas.
             Sono (1);
      }
}

//Quando nosso DLL é carregado, criamos uma thread no processo para criar um gancho.
//Precisamos fazer isso porque nosso DLL pode ser carregado antes que o processo
carregue o OpenGL.
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      Se(fdwReason==DL_Process_Attach){
```

```
            Reatethread(NULL,0,(lpthead_star_routine)injected_thread,
NULL,0,NULL);
      }

      Retornar a
verdade;}
```

A10 Terror Urbano OpenGL Chams

Veja Capítulo 5.4.

Um hacker chams do City Terror 4.3.4 revelou entidades através de paredes e mudou esses modelos para vermelho vivo. Ele funciona gancando a função OpenGL do jogo, glDrawElements, e desativando o teste de profundidade e a textura do OpenGL.

Isso é feito localizando a função glDrawElements na biblioteca OpenGL e criando uma caverna de código no início da função. Em uma caverna de código, examinamos o número de vértices associados a um elemento. Se exceder 500, chamamos o glDepthRange para limpar o plano de corte profundo e chamamos o glDepthFunc para desativar o teste profundo. Então, antes de definir a cor para vermelho usando o GCLOR, desativamos texturas e matrizes de cores e habilitamos materiais de cores. Caso contrário, chamamos essas mesmas funções para reativar o plano de recorte profundo, reativar o teste de profundidade e reativar a textura.

Este DLL deve ser injetado no processo de terror urbano para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>
#Incluindo <vetor>

HMODULE Openglhandele=NULL;

//Preenchemos dinamicamente os ponteiros de função das duas funções OpenGL
```

```
//Depois de injetar nossa DLL
void(__STDCal*GLDEPTHUNC)(unsigned int)=NULL;
void(__STDCal*GLDEPTHRGE)(double, double)=nul;

void(__stdcall*glColor4f)(float, float, float, float) = NULL;
void(__STDCal*Glennable)(unsigned int)=NULL;
void(__STDCal*GLDISLE)(unsigned int)=NULL;
void(__STDCal*GlenableClienteStation)(unsigned int)=NULL;
void(__stdcall*GLDISLEINTATE)(UnsigNET)=NULL;

unsigned char*hook_localização;

DWORD Ret_Adress=0;
DWORD old_protect;
Contagem DWORD = 0;

//Cave de código executada antes de chamar glDrawElements
__declspec(naked)void codecave(){
       //Primeiro, recuperamos os parâmetros de contato da chamada original.
      //Em seguida, buscamos o valor do parâmetro de contato, que especifica a
quantidade
      //Índice a ser apresentado
      _asm{
            Pushad
            mov eax, dword ptr ds: [esp+0x10]
            Contagem de metros, eax
            Rebote
            Pushad
      }

      //Se a contagem exceder 500, limparemos o plano de corte de profundidade e //
      definimos a função de profundidade para GL_ALWAYS
      //Em seguida, desativamos o array de cores e texturas e habilitamos o material
de cores antes de configurar
      //Cor vermelha
      Se (contar > 500){
            (*GLDEPTHRGE)(0,00,0);
            (*glDepthFunc)(0x207);

            (*GLDISLEITATE)(0x8078);
            (*GLDISBLILENTATE)(0x8076);
            (*glEnable)(0x0B57);
            (*glColor4f)(1,0 f, 0,6 f, 0,6 f, 1,0 f
      );}
      Caso contrário{
```

```
            //Caso contrário, restaure o plano de corte profundo ao valor padrão do
jogo e, em seguida,
            //Defina a função de profundidade para GL_LEQUAL e restaure a textura
            (*GLDEPTHRGE)(0,0,1,0);
            (*glDepthFunc)(0x203);

            (*GlenableClimentation Station)(0x8078);
            (*GlenableClimentation Station)(0x8076);
            (*GLDISBLE)(0x0B57);
            (*glColor4f)(1,0F, 1,0F, 1,0F, 1,0F);
      }

      //Por fim, restaure o comando original e salte de volta
      _asm{
            Rebote
            mov esi, dword ptr ds: [esi+0xA18]
            jmp Ret_Adress
      }
}

//Responsável por criar threads de injeção de gancho
void injectid_THREAD(){
      while(true){
            //Como o OpenGL é carregado dinamicamente no processo, nossos threads
precisam esperar
            //Até ver que o módulo OpenGL foi carregado. Se(Openglhandel=
            =NULL){
                  Opengll e=GetModuleHandle(L"opengl32.dll");
            }

            //Uma vez carregado, nós primeiro

            //Caverna de código acima
   if(Open GlDHANDLE!=NULE & GlDHANDLE ==NULE){GlDHANDLE =(void(__StDHANDL*)(
                                    unsigned)
int) GetProcadodres (openGLHandle, "glDepthFunc");
                  Gldepathrange=(void(__stdcal*)(double,
double) Gldepathrange (Opengldepathrange, "Gldepathrange");
                  glColor4f=(void (__stdcall*)(float, float, float,
float)) GetProcadress (Openglcolor4f, "glColor4f");
                  Glenanable=(void(__StDCal*)(unsigned
int)) Glenanable (OpenGlenanable, "Glenanable");
                  glDisable=(void(__StDSTAL*)(unsigned
int)) GetProcadernets (Opengldisable, "glDisable");
```

```
                    Glenglândalo=(void(__stdcal*)(unsigned int))
Glenglândalo (Openglândalo, "Glenglândalo");
                    Gldisable Climentation =(void(__stcal*)(unsigned int))
Gldisable Climentation (OpenGldisable Climentation, "Gldisable Climentation");

                    //Então encontramos a posição do glDrawElements e o deslocamos
para um comando fácil de ganhar
                    hook_location=(Sem símbolo
char*) GetProcaderse (Openglândle, "glDrawElements");
                    Hook_Localização+=0x16;

                    //Para ganchos, removemos a proteção da memória no código que
queremos escrever
        //Em seguida, defina o primeiro código de operação para E9 ou salte
                    //Usando a fórmula para calcular a posição: New_Localização-
Originação+5
                    //Por fim, como a primeira instrução original tem um total de 6
bytes, o NOP emite os últimos bytes restantes
                    Projeto Virtual* (void*)Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
                    *Hook_Localização=0xE9;
                    *(DWORD*)(hook_localização+1)=(DWORD)&codeave-
((DWORD)hook_localização+5);
                    *(hook_localização+5)=0x90;

                    //Como o OpenGL é carregado dinamicamente, precisamos calcular
dinamicamente o endereço de retorno
                     Ret_Adress=(DWORD) (hook_location+0x6);
            }

            //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
             //Isso permite que o processador agende outras tarefas.
            Sono (1);
      }
}

//Quando nosso DLL é carregado, criamos uma thread no processo para criar um gancho.
//Precisamos fazer isso porque nosso DLL pode ser carregado antes que o processo
carregue o OpenGL.
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      Se(fdwReason==DL_Process_Attach){
            Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL,
0, NULL);
```

```
        }

        Retornar a verdade;
}
```

A11 Cubo de Ataque Robô desencadeador

Veja Capítulo 5.5.

O robô gatilho que ataca o Cube 1.2.0.2 disparará a arma do jogador sempre que a linha cruzada cruza outro jogador.

Isso é feito através da função de gancho do jogo, que exibe uma etiqueta de nome quando você paira o mouse sobre o jogador. Sempre que o jogador paira sobre ele, nosso código caverna envia um evento de mouse para baixo para o jogo. Caso contrário, ele enviará um evento mouse para cima para interromper o acionamento.

Isso deve ser injetado no processo de cubo de ataque para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>

DWORD or_call_adress=0x4607C0;
DWORDORI_Jump_Adress=0x0040ADA2;

Entrada ={0};
DWORD edi_value=0;

//O programa executa a caverna de código para o qual saltará. A propriedade declspec
naked diz ao compilador para não adicionar nenhuma função
//Cabeças ao redor do código de montagem
__declspec(naked)void codecave(){
        //Os blocos Asm permitem que você escreva montagens puras
        //Neste caso, usamos isso para chamar a função à qual estamos vinculados e
salvar todos os registros
```

```
	//Depois que fizemos a chamada, movemos seu valor de retorno em eax para uma
variável
	_asm{
		Ligue para ori_Call_Adress
		Pushad
		mov edi_value, eax
	}

	//Se o resultado da chamada não for zero, então vemos um jogador

	//Crie um evento de mouse para simular o botão esquerdo do mouse sendo
pressionado e enviado para o jogo
	//Caso contrário, levante o botão do mouse para parar de atirar
	Se(edi_valor!=0){
		Entrada. Tipo = mouse de entrada;
		input.mi.dwFlags=MOUSEEVENTF_LEFTDOWN;
		SendInput (1, &input, sizeof (input));
	}
	Caso contrário{
		Entrada. Tipo = mouse de entrada;
		input.mi.dwFlags=mouseeventf_lefutup;
		SendInput (1, &input, sizeof (input));
	}

	//Restaurar o registro e retornar ao código original
	_asm{
		Rebote
		jmp ori_jump_adres
	}
}

//Quando nosso DLL é anexado, a proteção da memória é removida no código que queremos
escrever
//Em seguida, defina o primeiro código de operação para E9 ou salte
//Usando a fórmula para calcular a posição: New_Locação-
Localização original +5
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

	DWORD old_protect;
	unsigned char*Hook_Localização=(unsigned char*)0x0040AD9D;

	Se(fdwReason==DL_Process_Attach){
		Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
		*Hook_Localização=0xE9;
```

```
            *(DWORD*)(hook_localização+1)=(DWORD)&
codecavidade-((DWORD)hook_localização+5);
      }

      Retornar a
verdade;}
```

A.12 Cubo de Assalto

Abbott

Veja Capítulo 5.6.

Cubo de Assault 1.2.0.2 aimbot que mira automaticamente jogadores inimigos.

Ele funciona iterando a lista de inimigos e selecionando o inimigo mais próximo com base na distância euclidiana. Em seguida, use o compressor para calcular a curva e a inclinação necessárias para mirar no inimigo.

Isso deve ser injetado no processo de cubo de ataque para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>
#Incluindo <math.h>

//A função atan2f gera uma radiação. Para convertê-lo em um grau, precisamos do
valor do pi
#Definição M_PI 3.14159265358973323846

//Estrutura do jogador de cada jogador no jogo struct
Player{
      Caráter desconhecido 1
      [4]; Ponto flutuante x;
      y flutuante;
      z flutuante;
      Caráter desconhecido 2[
      0x30]; guia flutuante;
      espaçamento flutuante;
```

```
	char unknown3[0x2f0];
	int está morto;
};

//Nosso player
Player * Player = vazio;

//A função que calcula a distância euclídea entre dois pontos float
Euclidea_Diastance(float x, float y){
	Retorna a sqrtf(((x*x)+(y*y));
}

//Este thread contém todo o nosso código aimbot
void injectid_THREAD(){

	while(true){
		//Primeiro, capture a posição atual e o ângulo de visão do nosso player_
		offset=(DWORD*)(0x509B74);
		player* (player*)(*player_offset);

		 //Em seguida, obtenha o número de jogadores no jogo atual
		int*currente_Players=(int*)(0x50F500);

		//Essas variáveis serão usadas para manter o inimigo mais próximo de
		nós, float closest_player=-1.0f;
		float closest_yaw=0,0f;
		float closest_pitch=0,0f;

		//Viaje por todos os inimigos ativos
		for(int i=0;i i<*currente_players;i++){
			DWORD*enemy_list=(DWORD*)(0x50F4F8);
			DWORD*enemy_offset=(DWORD*)(*enemy_list+(i*4));
			Player*enemy=(Player*)(*enemy_offset);

		   //Certifique-se de que o inimigo seja eficaz e vivo
			Se (jogador! = vazio & inimigo! =vazio & ! Inimigo ->morre){

				//Calcule a posição absoluta do inimigo longe de nós para
garantir que nossos cálculos futuros sejam corretos e bem fundamentados.
				//Ao redor da origem
				float abspos_x=inimigo ->x-player->x;
				float abspos_y=inimigo->y-player->y;
				float abspos_z=inimigo->Z-jogador->z; //
				Calcule a distância entre nós e o inimigo
```

```
                        float Tempe_DISTANCE=Euclide_DISTANCE(abspos_x,//Se
abpos_y);               este for o inimigo mais próximo até agora, então calcule

Amirando-os com guia e inclinação
                        if(closest_player==-1.0f||temp_diaser<closest_
player){
                              closest_player=temp_diastação;

                              //Calculando o guinho
                              Ángulo de azimute flutuante xy=atan2f (abpos_y,
                              abpos_x); //Converter em grau
                              Curva flutuante = (flutuante) (ângulo azimutox Y*(180,0/
M_PI);
                              //Adicione 90, porque o jogo assume que o norte é
90 graus
                              closest_yaw=yaw+90;

                              //Calculando o tom
                              //Como o valor Z é muito limitado, selecione o
valor maior entre x e y para garantir que
                              //Não olhe diretamente para o ar ao se aproximar
inimigo
                                          Se (abpos_y<0){
                                    abpos_y*=-1;
                              }
                                          Se (abpos_y<5){
                                    Se (abpos_x<0){
                                          abpos_*=-1;
                                    }
                                    abpos_y=abpos_x;
                              }
                              Azimuto flutuante _z=atan2f (abspos_z, abspos_y
                              ); //Converter o valor para o grau closest_pitch
                              =(float)(azimuth_z*(180.0/
M_PI);
                        }
                  }
            }

            //Quando nosso loop termina, defina nosso guincho e inclinação aos
            valores mais próximos do player ->yaw=closest_yaw;
            player->pitch=closest_pitch;

            //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
```

```
            //Isso permite que o processador agende outras tarefas.
            Sono (1);
      }
}

//Quando nosso DLL é carregado, uma thread é criada no processo para processar o
código aimbot
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      Se(fdwReason==DL_Process_Attach){
            Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL
, 0, NULL);
      }

      Retornar a
verdade;}
```

O cubo de assalto A13 não tem recuo

Ver Capítulo 5.7.

Atacando hackers de Cube 1.2.0.2, eliminando todo o recuo ao disparar armas.

Isso é alcançado modificando o código do jogo responsável por definir o recuo do jogador. Ao alterar o comando final para alterar o valor do guinho do player e exibir um valor ignorado, o guinho do player nunca será modificado.

Isso deve ser injetado no processo de cubo de ataque para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>

//O novo código de operação para escrever o código do
jogo unsigned char new_bytes[3]={0xDD, 0xD8, 0x90};
```

```
//Quando nosso DLL é adicionado, primeiro desproteja a memória do jogo responsável
por aumentar o rebote.
//Em seguida, escreva nosso novo código de operação neste local de memória
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      DWORD old_protect;
      unsigned char*Hook_Localização=(unsigned char*)0x45bad;

      Se(fdwReason==DL_Process_Attach){
            Projeto Virtualítico ((void*)Hook_localizador, 3 página_Express_
Readwrate,&old_Projeto);
            for(int i=0; i<sizeof(new_bits); i++){*(hook_
                  localização+i)=new_bits[i];
            }
      }

      Retornar a
verdade;}
```

A.14 Cubo de Assalto

ESP

Veja Capítulo 5.9.

O ESP do Cubo Mágico de Ataque 1.2.0.2 exibe informações sobre jogadores inimigos acima de sua cabeça.

Ele funciona iterando a lista de inimigos e calculando a curva e a inclinação necessárias para mirar o inimigo usando o inimigo. Esta parte do código é retirada do código aimbot.

A diferença entre a curva e a inclinação calculadas e a curva e a curva e a inclinação do nosso jogador é então usada para derivar as coordenadas da tela do inimigo. Isso é feito adicionando a diferença multiplicada pelo fator de zoom ao meio da tela.

Isso deve ser injetado no processo de cubo de ataque para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

```
#include<Windows.h>
#Incluindo <math.h>

//A função atan2f gera uma radiação. Para convertê-lo em um grau, precisamos do
valor do pi
#Definição M_PI 3.14159265358973323846
//Número máximo de jogadores no Assault Cube
#Definir MAX_PLAYERS 32

//Estrutura do jogador de cada jogador no jogo struct
Player{
	Caráter desconhecido 1[4];
	Ponto flutuante x;
	y flutuante;
	z flutuante;
	Caráter desconhecido 2[0x30];
	guia flutuante;
	espaçamento flutuante;
	char unknown 3[0x 1dd];
	Nome do caractere[16];
};

//Nosso player
Player * Player = vazio;

DWORD Ret_Adress=0x0040BE83;
DWORD TEXT_ADRESS=0x419880;

//Nossa variável temporária para o código de texto impresso
é cave const char*text=";
const char*empty_text=";

DWORD x=0;
DWORD y=0;

//Lista de valores ESP calculados
DWORD x_valors[max_plays]={0}; DWORD
y_values[MAX_PLAYERS]={0}; char*NES[
MAX_PLAYERS]={NULL}; int*currente_
Players;
```

```
//Nossa caverna de código é responsável por imprimir o texto
__declspec(naked)void codecave(){
	Currente_Players=(int*)(0x50F500);

	Primeiro, recrie a função original à qual estamos vinculados, mas deixe o
texto em branco
	_asm{
		mov ecx, empty_text
		Texto de chamada _Adress
		Pushad
	}

	//Em seguida, percorra todos os jogadores atuais no jogo em um ciclo
	for (int i=1;i <*currente_players;i++){
		//Armazene a posição calculada da tela na variável temporária x=x_
		values[i];
		y=y_values[i];
		text=nomes[i];

		//Certifique-se de que nosso texto esteja na tela
		Se (x>2400||x<0|y<0|y>1800){
			text=";
		}

		x+=200;

		    //Chamar a função print text para exibir o texto
		_asm{
			mov ecx, texto
			Empurrar e soma
			Empurrar x
			Texto de chamada _Adress
			Adicionar esp, 8
		}
	}

	//Restaurar o registro e retornar ao código original
	_asm{
		Rebote
		jmp Ret_Adress
	}
}

//Este thread contém todo o código para calcular a localização da tela ESP
```

```
void injectid_THREAD(){
      while(true){
            //Primeiro, capture a posição atual e o ângulo de visão do nosso player_
            offset=(DWORD*) (0x509B74);
            player* (player*)(*player_offset);

             //Em seguida, obtenha o número de jogadores no jogo atual
            Currente_Players=(int*)(0x50F500);

            //Viaje por todos os inimigos ativos
            for (int i=1;i <*currente_players;i++){
                  DWORD*enemy_list=(DWORD*) (0x50F4F8);
                  DWORD*enemy_offset=(DWORD*) (*enemy_list+(i*4));
                  Player*enemy=(Player*)(*enemy_offset);

                  //Certifique-se de que o inimigo seja eficaz
                  Se(player!=NULL!=NULL){
                        //Calcule a posição absoluta do inimigo longe de nós para
garantir que nossos cálculos futuros sejam corretos e bem fundamentados.
                        //Ao redor da origem
                        float abspos_x=inimigo ->x-player->x;
                        float abspos_y=inimigo->y-player->y;
                        float abspos_z=inimigo->Z-jogador->z;

                        //Calculando o guinho
                        Ángulo de azimute flutuante xy=atan2f (abpos_y, abpos_x);
                        //Converter em grau
                        Cruzamento flutuante = (flutuante) (ângulo azimute xy*(180,0/M_PI));
                        //Adicione 90, porque o jogo assume que o norte é 90 graus

                        Caminho +=90;

                        //Calcule a diferença entre o nosso guinho atual e o
guinho calculado contra o inimigo
                        float yaw_dif=player->yaw-yaw;

                        //Se estivermos perto do limite do ângulo 275, nosso yaw_
dif será muito grande, causando erros no nosso texto.
                        //Para compensar isso, se yaw_dif for maior que 180,
subtraia yaw_dif de 360, porque nossa porta de visão nunca pode exibir 180 graus.
                        Se (yaw_dif>180){
                              yaw_dif=yaw_dif-360;}


                        Se (yaw_dif<-180){
```

```
                              yaw_dif=yaw_dif+360;
                        }

                        //Calcule nosso valor X (1200) multiplicando yaw_dif
pelo nível do fator de escalada ao centro da tela
                        x_values[i]=(DWORD) (1200+(yaw_dif*-30));

                        //Calculando o tom
                        //Como o valor Z é muito limitado, selecione o valor
maior entre x e y para garantir que
                        //Não olhe diretamente para o ar ao se aproximar do
inimigo
                        Se (abpos_y<0){
                              abpos_y*=-1;
                        }
                        Se (abpos_y<5){
                               Se (abpos_x<0){
                                     abpos_*=-1;
                              }
                              abpos_y=abpos_x;
                        }
                        Azimuto flutuante _z=atan2f (abspos_z, abspos_y);
                         //Converter valores em graus
                        Espaço flutuante = (flutuante) (azimutal_Z*(180,0/M_PI));

                           //O mesmo acima, mas o tom
                        float pitch_dif=player->pitch;

                        //Calcule nosso valor Y (900) multiplicando o pitch_dif
pelo fator de escala e adicionando-o verticalmente ao centro da tela
                        y_values[i]=(DWORD) (900+((pitch_dif)*25);

                        //Defina o nome como nome do inimigo
                        Nome [i]=Nome do inimigo ->;
                  }
            }

            //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
            //Isso permite que o processador agende outras tarefas.
            Sono (1);
      }
      }
```

```
//Quando nosso DLL é carregado, uma thread é criada no processo para processar o
código aimbot
//Em seguida, crie uma caverna de código para nossa função de texto impresso
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      DWORD old_protect;
      unsigned char*hook_localização=(unsigned char*)0x0040BE7E;

      Se(fdwReason==DL_Process_Attach){
            Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL
, 0, NULL);

            Projeto Virtualítico ((void*)Hook_Locasionismo, 5. Página_
Excessionismo, &ol_Projeto);
            *Hook_Localização=0xE9;
            *(DWORD*)(hook_localização+1)=(DWORD)&
codecavidade-((DWORD)hook_localização+5);
      }

      Retornar a
verdade;}
```

Cubo de Assalto A.15

Dohac

Veja Capítulo 5.10.

fusão

Como ponto de partida inicial do código multihack, combinamos todos os códigos-fonte diferentes dos cursos anteriores em um arquivo grande.

```
#include<Windows.h>#
include<math.h>
```

```
#Definição M_PI 3.14159265358973323846
#Definir MAX_PLAYERS 32

HMODULE Openglhandele=NULL;
void(__STDCal*GLDEPTHUNC) (unsigned int)=NULL; char*
OpenGL_Hook_LOK_LOCION sem símbolo;
DWORD OpenGl_Ret_Adress=0;

DWORD Trigerbot_or_Call_Adress=0x4607C0;
DWORD Trigerbot_orí_jump_Adress=0x0040ADA2;
Entrada ={0};
DWORD edi_value=0;

//Estrutura do jogador de cada jogador no jogo struct Player
{
	Caráter desconhecido 1[4];
	Ponto flutuante x;
	y flutuante;
	z flutuante;
	Caráter desconhecido 2[0x30];
	guia flutuante;
	espaçamento flutuante;
	char unknown 3[0x 1dd];
	Nome do caractere[16];
	Caráter desconhecido 4[0x103];
	int está morto;
};

//Nosso player
Player * Player = vazio;

DWORD ESP_ReT_Adress=0x0040BE83;
DWORD TEXT_Adress=0x419880;

//Nossa variável temporária para o código de texto impresso
é cave const char*text=";
const char*empty_text=";

DWORD x=0;
DWORD y=0;

//Lista de valores ESP calculados
DWORD x _valors[max_plays]={0}; DWORD
y_values[MAX_PLAYERS]={0}; char*NES[
MAX_PLAYERS]={NULL};
```

```
int*currente_Players;

DWORD old_protect;

//A função que calcula a distância euclídea entre dois pontos float
Euclidea_Diastance(float x, float y){
	Retorna a sqrtf(((x*x)+(y*y));
}

//Code cave, responsável por desativar os testes de profundidade do modelo
__declspec(naked)void opengl_codecave(){
	_asm{
		Pushad
	}

	(*glDepthFunc) (0x207);

	//Restaurar a instrução original e pular de volta
	_asm{
		Rebote
		mov esi, dword ptr ds: [esi+0xA18]
		jmp opengl_ret_address
	}
}

//Responsável por criar threads de injeção de gancho para OpenGL
void opengl_thread(){
	while(true){
		//Como o OpenGL é carregado dinamicamente no processo, nossos threads
precisam esperar
		//Até ver que o módulo OpenGL foi carregado. Se(Openglhandel
		==NULL){
			Openglle=GetModuleHandle (L"opengl32.dll");
		}

   if(Open GlDHANDLE!=NULE & GlDHANDLE ==NULE){GlDHANDLE =(void(__StDHANDL*)(
                                    unsigned)
int) GetProcadodres (openGLHandle, "glDepthFunc");

		//Então encontramos a posição do glDrawElements e o deslocamos
para um comando fácil de ganhar
		Opengl_Hook_Localização=(unsigned
char*)GetProcader* (openGLHandle, "Gldrawelementes");
		Opengl_hook_localização+=0x16;
```

```
                    //Para ganchos, removemos a proteção da memória no código
Espero escrever
        //Em seguida, defina o primeiro código de operação para E9 ou salte
                    //Usando a fórmula para calcular a posição: New_Localização-
Originação+5
                    //Por fim, como a primeira instrução original tem um total de 6
bytes, o NOP emite os últimos bytes restantes
                    Projeto Virtualítico ((void*)Opengl_Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
                    *OpenGl_Hook_Localização=0xE9;
                    *(DWORD*)(Opengl_hook_localização+1)=
(DWORD)&opengl_codecave-((DWORD)opengl_hook_location+5);
                    *(OpenGl_Hook_Localização+5)=0x90;

                    //Como o OpenGL é carregado dinamicamente, precisamos calcular
dinamicamente o endereço de retorno
                    opengl_ret_address=(DWORD) (OpenGl_Hook_Locação+0x6);
             }
             Caso contrário{
                    descanso;
             }

             //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
              //Isso permite que o processador agende outras tarefas.
             Sono (1);
      }
}

//Nosso código robô gatilho caverna
__declspec(naked)void triggerbot_codecave(){
      //Os blocos Asm permitem que você escreva montagens puras
      //Neste caso, usamos isso para chamar a função à qual estamos vinculados e
salvar todos os registros
      //Depois que fizemos a chamada, movemos seu valor de retorno em eax para uma
variável
      _asm{
             Chamar Triggerbot_ori_Call_Adress
             Pushad
             mov edi_value, eax
      }

      //Se o resultado da chamada não for zero, então vemos um jogador
```

```
	//Crie um evento de mouse para simular o botão esquerdo do mouse sendo
pressionado e enviado para o jogo
	//Caso contrário, levante o botão do mouse para parar de atirar
	Se(edi_valor!=0){
		Entrada. Tipo = mouse de entrada;
		input.mi.dwFlags=MOUSEEVENTF_LEFTDOWN;
		SendInput (1, &input, sizeof (input));
	}
	Caso contrário{
		Entrada. Tipo = mouse de entrada;
		input.mi.dwFlags=mouseeventf_lefutup;
		SendInput (1, &input, sizeof (input));
	}

	//Restaurar o registro e retornar ao código original
	_asm{
		Rebote
		jmp trigerbot_ori_jump_adres
	}
}

//Nossa caverna de código é responsável por imprimir o texto
__declspec(naked)void esp_codecave(){
	Currente_Players=(int*)(0x50F500);

	Primeiro, recrie a função original à qual estamos vinculados, mas deixe o
texto em branco
	_asm{
		mov ecx, empty_text
		Texto de chamada _Adress
		Pushad
	}

	//Em seguida, percorra todos os jogadores atuais no jogo em um ciclo
	for (int i=1;i <*currente_players;i++){
		//Armazene a posição calculada da tela na variável temporária x=x_
		values[i];
		y=y_values[i];
		text=nomes[i];

		//Certifique-se de que nosso texto esteja na tela
		Se(x>2400||x<0|y<0|y<0|y>1800){text="";

		}
```

```
                  //Chamar a função print text para exibir o texto
            _asm{
                  mov ecx, texto
                  Empurrar e soma
                  Empurrar x
                  Texto de chamada _Adress
                  Adicionar esp, 8
            }
      }

      //Restaurar o registro e retornar ao código original
      _asm{
            Rebote
            jmp ESP_Ret_Adress
      }
}

//Este thread contém todo o nosso código aimbot e ESP
void AIMBOT_THEAD(){

      while(true){
            //Primeiro, capture a posição atual e o ângulo de visão do nosso player_
            offset=(DWORD*) (0x509B74);
            player* (player*)(*player_offset);

             //Em seguida, obtenha o número de jogadores no jogo atual
            int*currente_Players=(int*)(0x50F500);

            //Essas variáveis serão usadas para manter o inimigo mais próximo de
            nós, float closest_player=-1.0f;
            float closest_yaw=0,0f;
            float closest_pitch=0,0f;

            //Viaje por todos os inimigos ativos
            for(int i=0;i i<*currente_players;i++){
                  DWORD*enemy_list=(DWORD*) (0x50F4F8);
                  DWORD*enemy_offset=(DWORD*) (*enemy_list+(i*4)); Player*
                  enemy=(Player*)(*enemy_offset);

               //Certifique-se de que o inimigo seja eficaz e vivo
                  Se(player!=NULL!=NULL){

                        //Calcule a posição absoluta do inimigo longe de nós para
garantir que nossos cálculos futuros sejam corretos e bem fundamentados.
                        //Ao redor da origem
```

```
                        float abspos_x=inimigo ->x-player->x;
                        float abspos_y=inimigo->y-player->y;
                        float abspos_z=inimigo->Z-jogador->z;

                        //Calcule a distância entre nós e o inimigo
                        float Tempe_DISTANCE=Euclide_DISTANCE(abspos_x,//Se
abpos_y);               este for o inimigo mais próximo até agora, calcule

Amirando-os com guia e inclinação

                        Ángulo de azimute flutuante xy=atan2f (abpos_y, abpos_x); Cruzamento
                        flutuante = (flutuante) (ângulo azimute xy*(180,0/M_PI));

                        Caminho +=90;

                        //Calcule a diferença entre o nosso guinho atual e o
guinho calculado contra o inimigo
                        float yaw_dif=player->yaw-yaw;

                        //Se estivermos perto do limite do ângulo 275, nosso yaw_
dif será muito grande, causando erros no nosso texto.
                        //Para compensar isso, se yaw_dif for maior que 180,
subtraia yaw_dif de 360, porque nossa porta de visão nunca pode exibir 180 graus.
                        Se (yaw_dif>180){
                              yaw_dif=yaw_dif-360;
                        }

                        Se (yaw_dif<-180){
                              yaw_dif=yaw_dif+360;
                        }

                        //Calcule nosso valor X (1200) multiplicando yaw_dif
pelo nível do fator de escalada ao centro da tela
                        x_values[i]=(DWORD) (1200+(yaw_dif*-30));

                        //Calculando o tom
                        //Como o valor Z é muito limitado, escolha o valor maior
entre x e y para garantir que
                        //Não olhe diretamente para o ar ao se aproximar do
inimigo
                        Se (abpos_y<0){abpos_
                              y*=-1;
                        }
                        Se(abpos_y<5){if(abpos_x
                              <0){
                                    abpos_*=-1;
```

```
                              }
                              abpos_y=abpos_x;
                        }
                        Azimuto flutuante _z=atan2f (abspos_z, abspos_y); Espaço
                        flutuante = (flutuante) (azimutal_Z*(180,0/M_PI));
                           //O mesmo acima, mas o tom
                        float pitch_dif=player->pitch;

                        //Calcule nosso valor Y (900) multiplicando o pitch_dif
pelo fator de escala e adicionando-o verticalmente ao centro da tela
                        y_values[i]=(DWORD) (900+((pitch_dif)*25);

                        //Defina o nome como nome do inimigo
                        Nome [i]=Nome do inimigo ->;

                        if((closest_player==-1.0f||temp_distance<closest_
player)&&! Espero->dead){
                              closest_player=temp_diastação;
                              closest_yaw=guia;
                              closest_pitch=pitch;
                        }
                  }
            }

            //Quando nosso loop termina, defina nosso guincho e inclinação aos
            valores mais próximos do player ->yaw=closest_yaw;
            player->pitch=closest_pitch;

            //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
             //Isso permite que o processador agende outras tarefas.
            Sono (1);
      }
}

//Quando nosso DLL é carregado, criamos uma thread no processo para criar um gancho.
//Precisamos fazer isso porque nosso DLL pode ser carregado antes que o processo
carregue o OpenGL.
//Também criamos threads aimbot e ESP e vinculam a localização do TiggerBOT e do
texto impresso
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      unsigned char*Trigerbot_Hook_Location=(unsigned char*)0x0040AD9D;
      unsigned char*esp_hook_location=(unsigned char*)0x0040BE7E;
```

```
      Se (fdwReason==DL_Process_Attach){CreateThread(NULL, 0, (lpthread_
            star_routine)opengl_thread,
Vazio, 0, vazio);
            Reatethread (NULL, 0, (lpthead_star_routine) aimbot_thread,
NULL, 0, NULL);

            Projeto Virtualícola ((void*)Trigerbot_Hook_Locasion,5,
página_Express_Readwrite, &old_protect);
            *TrigerBot_Hook_Localização=0xE9;
            *(DWORD*)(trigerbot_hook_localização+1)=
(DWORD)&TrigerBot_Codecave-((DWORD)TrigerBot_Hook_Locação+5);

            Projeto Virtualítico ((void*)ESP_Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
            *ESP_Hook_Localização=0xE9;
            *(DWORD*)(ESP_hook_localização+1)=(DWORD)&ESP_codificação-((
DWORD)ESP_hook_localização+5);
      }

      Retornar a
verdade;}
```

A primeira reconstrução

Nossa primeira refactorização do código multihack, na qual dividimos a função TrigerBot em sua própria classe.

Título/Trigerbot.H

```
#pragma uma vez

#include<Windows.h>

Robô semelhante a gatilho{
Privado:
      Entrada ={0};
Público:
      Trigerbot ();
      void Expectation (int Islooking
Ingenation); };
```

Fonte/Robótico.cpp

```
#include<Windows.h>

#Incluindo "Trigerbot.H"

Trigerbot:: Trigerbot(){
	Entrar ={0};
}

//Se o iSlookingatemy não é zero, então estamos procurando um jogador
//Crie um evento de mouse para simular o botão esquerdo do mouse sendo pressionado e
enviado para o jogo
//Caso contrário, levante o botão do mouse para parar de acionar
o void TraigGerbot:: Expressway (int IslookIngenation){
	Se(iSlookingatemy! =0){
		Entrada. Tipo = mouse de entrada;
		input.mi.dwFlags=MOUSEEVENTF_LEFTDOWN;
		SendInput (1, &input, sizeof (input));
	}
	Caso contrário{
		Entrada. Tipo = mouse de entrada;
		input.mi.dwFlags=mouseeventf_lefutup;
		SendInput (1, &input, sizeof (input));
	}
}
```

Fonte/anfitriã.cp

```
#include<Windows.h>
#Incluindo <math.h>

#Incluindo "Trigerbot.H"

#Definição M_PI 3.14159265358973323846
#Definir MAX_PLAYERS 32

//Nossa classe de gatilho
Robótico* Robótico* Robótica acionada;

HMODULE Openglhandele=NULL;
void(__STDCal*GLDEPTHUNC) (unsigned int)=NULL; char
*OpenGL_Hook_LOK_LOCION sem símbolo;
DWORD OpenGl_Ret_Adress=0;

DWORD Trigerbot_orí_Call_Adress=0x4607C0;
```

```
DWORD Trigerbot_orí_jump_Adress=0x0040ADA2;
DWORD edi_value=0;

//Estrutura do jogador de cada jogador no jogo
Player estrutural{
	Caráter desconhecido 1[4];
	Ponto flutuante x;
	y flutuante;
	z flutuante;
	Caráter desconhecido 2[0x30];
	guia flutuante;
	espaçamento flutuante;
	char unknown 3[0x 1dd];
	Nome do caractere[16];
	Caráter desconhecido 4[0x103];
	int está morto;
};

//Nosso player
Player * Player = vazio;

DWORD ESP_ReT_Adress=0x0040BE83;
DWORD TEXT_Adress=0x419880;

//Variáveis temporárias em nossa caverna de código de texto impresso
const char*text="";
const char*empty_text=";

DWORD x=0;
DWORD y=0;

//Lista de valores ESP calculados
DWORD x_values[MAX_PLAYERS]={0};
DWORD y_values[MAX_PLAYERS]={0};
char*NES[MAX_PLAYERS]={NULL};

int*currente_Players;

DWORD old_protect;

//A função que calcula a distância euclídea entre dois pontos float
Euclidea_Diastance(float x, float y){
	Retornar sqrtf(((x*x)+(y*y));}
```

```
//Nossa caverna de código glDrawElements é responsável pelo nosso wallhack
__declspec(naked)void opengl_codecave(){
	_asm{
		Pushad
	}

	(*glDepthFunc)(0x207);

	//Por fim, restaure o comando original e salte de volta
	_asm{
		Rebote
		mov esi, dword ptr ds: [esi+0xA18]
		jmp OpenGl_Ret_Adress
	}
}

//Responsável por criar threads de injeção para ganchos OpenGL
void opengl_thread(){
	while(true){
		//Como o OpenGL é carregado dinamicamente no processo, nossos threads
precisam esperar
		//Até ver que o módulo OpenGL foi carregado. Se(Openglhandel
		==NULL){
			Openglle=GetModuleHandle(L"opengl32.dll");
		}

   if(Open GlDHANDLE!=NULE & GlDHANDLE ==NULE){GlDHANDLE =(void(__StDHANDL*)(
                                    unsigned)
int) GetProcadodres (openGLHandle, "glDepthFunc");

			//Então encontramos a posição do glDrawElements e o deslocamos
para um comando fácil de ganhar
			Opengl_hook_localização=(unsigned)
char*) GetProcaderse (Openglândle, "glDrawElements");
			Opengl_hook_localização+=0x16;

			//Para ganchos, removemos a proteção da memória no código que
queremos escrever
        //Em seguida, defina o primeiro código de operação para E9 ou salte
			//Usando a fórmula para calcular a posição: New_Localização-
Originação+5
			//Por fim, como a primeira instrução original tem um total de 6
bytes, o NOP emite os últimos bytes restantes
			Projeto Virtual_Projeto ((void*)OpenGL_Hook_
Locasionismo, 5. Page_Exploração, &old_Projeto);
```

```
                    *OpenGl_Hook_Localização=0xE9;
                    *(DWORD*)(Opengl_hook_localização+1)=
(DWORD)&opengl_codecave-((DWORD)opengl_hook_location+5);
                    *(OpenGl_Hook_Localização+5)=0x90;

                    //Como o OpenGL é carregado dinamicamente, precisamos
calcular dinamicamente o endereço de retorno
                    opengl_ret_address=(DWORD) (OpenGl_Hook_Locação+0x6);}

             Caso contrário{
                    descanso;
             }

             //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
             //Isso permite que o processador agende outras tarefas.
             Sono (1);
      }
}

//Nossa Caverna de Código TrigerBot
__declspec(naked)void Trigerberbot_Codecave(){
      //Restaurar a chamada original e obter o valor do edi que contém se estamos
verificando o jogador
      _asm{
             Chamar Triggerbot_ori_Call_Adress
             Pushad
             mov edi_value, eax
      }

      //Envie esta informação para a instância do TrigerBot para determinar o que
fazer
      Trigerbot->Experiência (edi_valor);

      //Restaurar o registro e pular de volta para o CODE_ASM{

             Rebote
             jmp trigerbot_ori_jump_adres
      }
}

//Somos responsáveis por imprimir text_declspec(
naked)void esp_codecave(){
      Currente_Players=(int*)(0x50F500);
```

```
	Primeiro, recrie a função original à qual estamos vinculados, mas deixe o
texto em branco
	_asm{
		mov ecx, empty_text
		Texto de chamada _Adress
		Pushad
	}

	//Em seguida, todos os jogadores atuais no jogo são relocados
	for (int i=1;i <*currente_players;i++){
		//Armazene a posição calculada da tela na variável temporária x=x_
		values[i];
		y=y_values[i];
		text=nomes[i];

		//Certifique-se de que nosso texto esteja na tela
		Se (x>2400||x<0|y<0|y>1800){
			text=";
		}

		     //Chamar a função print text para exibir o texto
		_asm{
			mov ecx, texto
			Empurrar e soma
			Empurrar x
			Texto de chamada _Adress
			Adicionar esp, 8
		}
	}

	//Restaurar o registro e retornar ao código original
	_asm{
		Rebote
		jmp ESP_Ret_Adress
	}
}

//Este thread contém todo o nosso código aimbot e ESP
void AIMBOT_THEAD(){

	while(true){
		//Primeiro, capture a posição atual e o ângulo de visão do nosso player_
		offset=(DWORD*) (0x509B74);
		player* (player*)(*player_offset);
```

```
            //Em seguida, obtenha o número de jogadores no jogo atual
            int*currente_Players=(int*)(0x50F500);

            //Essas variáveis serão usadas para manter o inimigo mais próximo de
            nós, float closest_player=-1.0f;
            float closest_yaw=0,0f;
            float closest_pitch=0,0f;

            //Viaje por todos os inimigos ativos
            for(int i=0;i i<*currente_players;i++){
                  DWORD*enemy_list=(DWORD*) (0x50F4F8);
                  DWORD*enemy_offset=(DWORD*) (*enemy_list+(i*4)); Player*
                  enemy=(Player*)(*enemy_offset);

               //Certifique-se de que o inimigo seja eficaz e vivo
                  Se(player!=NULL!=NULL){

                        //Calcule a posição absoluta do inimigo longe de nós para
garantir que nossos cálculos futuros sejam corretos e bem fundamentados.
                        //Ao redor da origem
                        float abspos_x=inimigo ->x-player->x;
                        float abspos_y=inimigo->y-player->y;
                        float abspos_z=inimigo->Z-jogador->z;

                        //Calcule a distância entre nós e o inimigo
                        Temperatura de float = Euclide_Diance (abspos_x, abspos_
y);
                        //Se este for o inimigo mais próximo até agora, calcule o
guinho e mire nele

                        Ángulo de azimute flutuante xy=atan2f (abpos_y, abpos_x); Cruzamento
                        flutuante = (flutuante) (ângulo azimute xy*(180,0/M_PI));

                        Caminho +=90;

                        //Calcule a diferença entre nosso guinho atual e o guinho
calculado contra o inimigo
                        float yaw_dif=player->yaw-yaw;

                        //Se estivermos perto do limite do ângulo 275, nosso yaw_
dif será muito grande, fazendo com que nosso texto pareça incorreto.
                        //Para compensar isso, se yaw_dif for maior que 180,
subtraia yaw_dif de 360, porque nossa porta de visão nunca pode exibir 180 graus.
                        Se (yaw_dif>180){
                              yaw_dif=yaw_dif-360;}
```

```
                        Se (yaw_dif<-180){
                              yaw_dif=yaw_dif+360;
                        }

                        //Calcule nosso valor X (1200) multiplicando yaw_dif
pelo nível do fator de escalada ao centro da tela
                        x_values[i]=(DWORD) (1200+(yaw_dif*-30));

                        //Calculando o tom
                        //Como o valor Z é muito limitado, escolha o valor maior
entre x e y para garantir que
                        //Não olhe diretamente para o ar ao se aproximar do
inimigo
                        Se (abpos_y<0){
                              abpos_y*=-1;
                        }
                        Se (abpos_y<5){
                               Se (abpos_x<0){
                                    abpos_*=-1;
                              }
                              abpos_y=abpos_x;
                        }
                        Azimuto flutuante _z=atan2f (abspos_z, abspos_y); Espaço
                        flutuante = (flutuante) (azimutal_Z*(180,0/M_PI));
                           //O mesmo acima, mas o tom
                        float pitch_dif=player->pitch;

                        //Calcule nosso valor Y (900) multiplicando o pitch_dif
pelo fator de escala e adicionando-o verticalmente ao centro da tela
                        y_values[i]=(DWORD) (900+((pitch_dif)*25);

                        //Defina o nome como nome do inimigo
                        Nome [i]=Nome do inimigo ->;

                        if((closest_player==-1.0f||temp_distance<closest_
player)&&! Espero->dead){
                              closest_player=temp_diastação;
                              closest_yaw=guia;
                              closest_pitch=pitch;
                        }
                  }
            }

//Quando nosso ciclo termina, defina nosso guinho e inclinação aos valores mais próximos.
```

```
            Player -> Jame = Jame mais próximo;
            player->pitch=closest_pitch;

            //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
             //Isso permite que o processador agende outras tarefas.
            Sono (1);
      }
}

//Quando nosso DLL é carregado, criamos uma thread no processo para criar um gancho
. //Precisamos fazer isso porque nosso DLL pode ser carregado antes que o processo
carregue o OpenGL.
//Também criamos threads aimbot e ESP e vinculam a localização do TiggerBOT e do
texto impresso
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      unsigned char*Trigerbot_Hook_Location=(unsigned char*)0x0040AD9D;
      unsigned char*esp_hook_location=(unsigned char*)0x0040BE7E;

      Se(fdwReason==DL_Process_Attach){
            //Crie nosso robô gatilho
            Tiggerbot=novo Tiggerbot();

            Creathread (NULL, 0, (lpthead_star_routine) opengl_thread, NULL, 0,
NULL);
            Reatethread (NULL, 0, (lpthead_star_routine) aimbot_thread, NULL, 0
, NULL);

            Projeto Virtualícola ((void*)Trigerbot_Hook_Locasion,5,
página_Express_Readwrite, &old_protect);
            *TrigerBot_Hook_Localização=0xE9;
            *(DWORD*)(trigerbot_hook_localização+1)=
(DWORD)&TrigerBot_Codecave-((DWORD)TrigerBot_Hook_Locação+5);

            Projeto Virtualítico ((void*)ESP_Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
            *ESP_Hook_Localização=0xE9;
            *(DWORD*)(ESP_hook_localização+1)=(DWORD)&ESP_codificação
-((DWORD)ESP_hook_localização+5);
      }
      else if (fdwReason==DL_Process_Detach){
            delete Trigerbot;;;
      }
```

```
        Retornar a verdade;
}
```

Reconstrução concluída

O código completo do multihack após a refactorização ser concluída.
Header/Trigerbot.H- Sem mudanças em relação à primeira reconstrução

Cabeça/Constante.H

```
#pragma uma vez

#include<Windows.h>

DWORD Trigerbot_or_Call_Adress=0x4607C0; DWORD
Trigerbot_orí_jump_Adress=0x0040ADA2;

DWORD ESP_ReT_Adress=0x0040BE83;
DWORD TEXT_Adress=0x419880; const
char*empty_text="";
```

Título/Player Geometry.H

```
#pragma uma vez

#include<Windows.h>

#Definição M_PI 3.14159265358973323846
#Definir MAX_PLAYERS 32

//Estrutura do jogador de cada jogador no jogo struct
Player{
        Caráter desconhecido 1
        [4]; Ponto flutuante x;
        y flutuante;
        z flutuante;
        Caráter desconhecido 2[
        0x30]; guia flutuante;
        espaçamento flutuante;
        char unknown 3[0x 1dd];
```

```
        Nome do caractere[16];
        Caráter desconhecido 4[0x103];
        int está morto;
};

Qualquer player Geometry{
Privado:
        DWORD Play_Offset_Adress;
        DWORD Enemy_list_Adress;
        DWORD Currente_Players_Adress;

        closest_yaw flutuante;
        closest_pitch flutuante;

        Jogadores * Jogadores;

        float Euclidea_Dostance (float, float);
Público:
        DWORD x_values[ MAX _PLAYERS] ={0}; DWORD y_
        values[ MAX _PLAYERS] ={0}; char *NES[ MAX _
        PLAYERS]={NULL};

        int*currente_Players;

        Geometria do jogador (DWORD, DWORD, DWORD);
        void update();
        void se_Player_VIEW();
};
```

Source/Trigerbot.cpp-Nenhuma mudança em relação à primeira reconstrução Source/PlayerGeometry.cpp

```
#include<Windows.h>
#Incluindo <math.h>

#Incluindo "PlayerGeometry.h"

Player Geometry:: Player Geometry (DWORD P_Adress, DWORD E_Adress, DWORD Cp
_Adress){
        Player_Offset_Adress=P_Adress; enemy_
        list_address=e_address; Currente_
        Players_Adress=cp_Adress;
}
```

```
//Função que calcula a distância euclídea entre dois pontos float
PlayerGeometry::: Euclidea_distance(float x, float y){
      Retorna a sqrtf(((x*x)+(y*y));
}

void Player Geometry:: update(){
      //Primeiro, pegue a posição atual e o ângulo de visão do nosso player_offset=(
      DWORD*) (player_offset);
      player* (player*)(*player_offset);

      //Em seguida, obtenha o número de jogadores no jogo atual
      Currente_Players=(int*)(0x50F500);

      float closest_player=-1.0f;
      closest_yaw=0,0f;
      closest_pitch=0,0f;

      //Viaje por todos os inimigos ativos
      for (int i=0;i i<*currente_players;i++){
            DWORD*enemy_list=(DWORD*) (0x50F4F8);
            DWORD*enemy_offset=(DWORD*) (*enemy_list+(i*4)); Player
            *enemy=(Player*)(*enemy_offset);

            //Certifique-se de que o inimigo seja eficaz e vivo
            Se(player!=NULL!=NULL){
                  //Calcule a posição absoluta do inimigo longe de nós para
garantir que nossos cálculos futuros sejam corretos e bem fundamentados.
                  //Ao redor da origem
                  float abspos_x=inimigo ->x-player->x;
                  float abspos_y=inimigo->y-player->y;
                  float abspos_z=inimigo->Z-jogador->z;

                   //Calcule a distância entre nós e o inimigo
                  Temperatura Float_DISTANCE=Euclide_DISTANCE(abspos_x,
abpos_y);
                  //Se este for o inimigo mais próximo até agora, calcule o guinho
e mire nele

                  Ángulo de azimute flutuante xy=atan2f (abpos_y, abpos_x); Cruzamento
                  flutuante = (flutuante) (ângulo azimute xy*(180,0/M_PI));

                  Caminho +=90;

                  //Calcule a diferença entre o nosso guinho atual e o guinho
calculado contra o inimigo
```

```
                    float yaw_dif=player->yaw-yaw;

                    //Se estivermos perto do limite do ângulo 275, nosso yaw_dif
será muito grande, causando erros no nosso texto.
                    //Para compensar isso, se yaw_dif for maior que 180, subtraia
yaw_dif de 360, porque nossa porta de visão nunca pode exibir 180 graus.
                    Se (yaw_dif>180){
                          yaw_dif=yaw_dif-360;
                    }

                    Se (yaw_dif<-180){
                          yaw_dif=yaw_dif+360;
                    }

                    //Calcule nosso valor X (1200) multiplicando yaw_dif pelo nível
do fator de escalada ao centro da tela
                    x_values[i]=(DWORD) (1200+(yaw_dif*-30));

                    //Calculando o tom
                    //Como o valor Z é muito limitado, escolha o valor maior entre x e
y para garantir que
                    //Não olhe diretamente para o ar ao se aproximar do inimigo,
                    se (abspos_y<0){
                          abpos_y*=-1;
                    }
                    Se (abpos_y<5){
                          Se (abpos_x<0){
                                abpos_*=-1;
                          }
                          abpos_y=abpos_x;
                    }
                    Azimuto flutuante _z=atan2f (abspos_z, abspos_y); Espaço
                    flutuante = (flutuante) (azimutal_Z*(180,0/M_PI));
                    //O mesmo acima, mas o tom
                          float pitch_dif=player->pitch;

                    //Calcule nosso valor Y (900) multiplicando o pitch_dif pelo
fator de escala e adicionando-o verticalmente ao centro da tela
                    y_values[i]=(DWORD) (900+((pitch_dif)*25);

                    //Defina o nome como nome do inimigo
                    Nome [i]=Nome do inimigo ->;

                    if((closest_player==-1.0f||temp_distance<
closest_player)&&! Espero->dead){
```

```
                        closest_player=temp_diastação;
                        closest_yaw=guia;
                        closest_pitch=pitch;
                  }
            }
      }
}

void player Geometry::: set_player_view
      (){player->yaw=closest_yaw; player
      ->player=Closest_Player;
}
```

Fonte/host.cpp

```
#include<Windows.h>

#Incluindo "constants.h"
#Incluindo "Trigerbot.H"
#Incluindo "PlayerGeometry.h"

Robótico* Robótico* Robótica acionada
Geometria do jogador *Geometria do jogador;

HMODULE Openglhandele=NULL;
void(__STDCal*GLDEPTHUNC) (unsigned int)=NULL;
DWORD OpenGl_Ret_Adress=0;

DWORD edi_value=0;

//Variáveis temporárias em nossa caverna de código de texto impresso
const char*text="";

Dupla palavra x;
Dupla palavra y;

DWORD old_protect;

//A caverna de código responsável por desativar testes de
profundidade no modelo _declspec(naked) void opengl_codecave(){
      _asm{
            empur
            rar
      }
```

```
	(*glDepthFunc)(0x207);

	//Por fim, restaure o comando original e salte de volta
	_asm{
		Rebote
		mov esi, dword ptr ds:[esi+0xA18]
		jmp OpenGl_Ret_Adress
	}
}

//Código Cave é responsável pelo nosso robô de gatilho
__declspec(naked)void triggerbot_codecave(){
	//Os blocos Asm permitem que você escreva montagens puras
	//Neste caso, usamos isso para chamar a função à qual estamos vinculados e
salvar todos os registros
	//Depois que fizemos a chamada, movemos seu valor de retorno em eax para uma
variável
	_asm{
		Chamar Triggerbot_ori_Call_Adress
		Pushad
		mov edi_value, eax
	}

	Trigerbot->Experiência(edi_valor);

	//Restaurar o registro e retornar ao código original
	_asm{
		Rebote
		jmp trigerbot_ori_jump_adres
	}
}

//Nossa caverna de código é responsável por imprimir o texto
__declspec(naked)void text_codecave(){
	Primeiro, recrie a função original à qual estamos vinculados, mas deixe o
texto em branco
	_asm{
		mov ecx, empty_text
		Texto de chamada _Adress
		Pushad
	}

	//Em seguida, percorra todos os jogadores atuais no jogo em um ciclo
	Para (int i=1; i<*playerGeometry->current_players;i++){
		//Armazenar a localização calculada da tela em uma variável temporária
```

```
            x=playerGeometry->x_values[i];
            y=playerGeometry->y_values[i];
            text=player geometry->Nomes[i];

            //Certifique-se de que nosso texto esteja na tela
            Se (x>2400||x<0|y<0|y>1800){
                  text=";
            }

                //Chamar a função print text para exibir o texto
            _asm{
                  mov ecx, texto
                  Empurrar e soma
                  Empurrar x
                  Texto de chamada _Adress
                  Adicionar esp, 8
            }
      }

      //Restaurar o registro e retornar ao código original
      _asm{
            Rebote
            jmp ESP_Ret_Adress
      }
}

//Este thread contém todos os nossos códigos aimbot, ESP e OpenGL
void injectid_THREAD(){

      while(true){
            Se(Openglhandel==NULL){
                  Openglle=GetModuleHandle (L"opengl32.dll");
            }

  if(Open GlDHANDLE!=NULE & GlDHANDLE ==NULE){GlDHANDLE =(void(__StDHANDL*)(
                                    unsigned)
int) GetProcadodres (openGLHandle, "glDepthFunc");

                  //Então encontramos a posição do glDrawElements e o
deslocamos para um comando fácil de ganhar
                  unsigned char*OpenGl_Hook_Lokadeion=(unsigned
char*) getProcadernets ("
                  Opengl_hook_localização+=0x16;
```

```
                    //Para ganchos, removemos a proteção da memória no código
Espero escrever
        //Em seguida, defina o primeiro código de operação para E9 ou salte
                    //Usando a fórmula para calcular a posição: New_Localização-
Originação+5
                    //Por fim, como a primeira instrução original tem um total de 6
bytes, o NOP emite os últimos bytes restantes
                    Projeto Virtualítico ((void*)Opengl_Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
                    *OpenGl_Hook_Localização=0xE9;
                    *(DWORD*)(Opengl_hook_localização+1)=
(DWORD)&opengl_codecave-((DWORD)opengl_hook_location+5);
                    *(OpenGl_Hook_Localização+5)=0x90;

                    //Como o OpenGL é carregado dinamicamente, precisamos calcular
dinamicamente o endereço de retorno
                    opengl_ret_address=(DWORD) (OpenGl_Hook_Locação+0x6);
             }

             Player Geometry->update();
             playerGeometry->set_player_view();

             //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
              //Isso permite que o processador agende outras tarefas.
             Sono (1);
      }
}

//Quando nosso DLL é carregado, criamos uma thread no processo para criar um gancho
. //Precisamos fazer isso porque nosso DLL pode ser carregado antes que o processo
carregue o OpenGL.
//Também criamos threads aimbot e ESP e vinculam a localização do TiggerBOT e do
texto impresso
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      unsigned char*Trigerbot_Hook_Location=(unsigned char*)0x0040AD9D;
      unsigned char*text_hook_localização=(unsigned char*)0x0040BE7E;

      Se(fdwReason==DL_Process_Attach){
            Tiggerbot=novo Tiggerbot();
            Player Geometry = Novo Player Geometry (0x 509b 74, 0x 50f 4f
8, 0x 50f 500);
```

```
            Reatethread (NULL, 0, (lpthead_star_routine) injected_thread,
NULL, 0, NULL);

            Projeto Virtualícola ((void*)Trigerbot_Hook_Locasion,5,
página_Express_Readwrite, &old_protect);
            *TrigerBot_Hook_Localização=0xE9;
            *(DWORD*)(trigerbot_hook_localização+1)=
(DWORD)&TrigerBot_Codecave-((DWORD)TrigerBot_Hook_Locação+5);

            Projeto Virtualítico ((void*)Text_Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
            *text_hook_localização=0xE9;
            *(DWORD*)(text_hook_localização+1)=(DWORD)&text_codificação-((
DWORD)text_hook_localização+5);
      }
      else if (fdwReason==DL_Process_Detach){
            delete Trigerbot;;;
            Excluir player Geometry
      ;}

      Retornar a
verdade;}
```

Concluído

Os hackers múltiplos que atacam o Cube 1.2.0.2 incluem as seguintes funções:

- Hacker de parede
- ESP
- Abbott
- Robô desencadeador

Ele também tem um menu interativo que permite que esses recursos sejam ativados e desativados. Use as setas para cima e para baixo para alterar a seleção e use a função de alternância das setas para a esquerda e para a direita

Isso deve ser injetado no processo de cubo de ataque para funcionar. Uma maneira é usar um injetor DLL. Outra maneira é ativar AppInit_DLLs no registro.

Header/Player Geometry.H- inalterado com "Reconstruction Complete"

Header/Trigerbot.H-- inalterado com "Reconstrução Completa"

Header/constants.h- e "reconfigurado completo" Header/
Menu.h permanece inalterado

```
#pragma uma vez

#Definir WALLHACK 0
#Definir ESP 1
#Definir AIMBOT 2
#Definir Trigerbot 3

#Definir MAX_ITEMS 4

Menu de classe{
Privado:
      const char on_text[5]={0xc, 0x30, 'O', 'N', 0};
      const char off_text[6]={0xc, 0x33, 'O', 'F', 'F', 'F', '0};
Público:
      Posição do cursor int;

      const char*items[MAX_ITEMS]={"Wallhack", "ESP", "Aimbot", "
Trigerberbot"};
      bool ITEM_EBLEDs[MAX_ITEMS]={false}; const
      char*cursor=">";


      Menu ();
      void handle_input(); const
      char*get_Stado (int);
};
```

Source/Trigerbot.cpp-inalterado com "reconfiguração completa"
Source/PlayerGeometry.cpp- com "reconfiguração completa"
inalterado com "reconfiguração completa" Source/Menu.cpp

```
#include<Windows.h>

#Incluindo "Menu.h"

Menu:: Menu (){
      Cursor_Posição=0;}
```

```
void Menu:: handle_input(){

	if(GetaseNCKEYTE(VK_DON)&1){
		Posição do cursor ++;
	}
	else if(GetaseNCK_UP)&1){
		Posição do cursor-;
	}
	else if((VK_LEFT)&1)||(VK_LEFT)(VK_LEFT)&1)){

		item_enabled[Cursor_Posição]=! item_enabled[Cursor_Posição];}


	Se(Cursor_Posição<0){
		Cursor_Posição=3;
	}
	Esse if(Cursor_Posição>3){
		Cursor_Posição=0;
	}
}

const char*Menu:: get_STAR_STAR_STAR_STAR_STAR_STAR_
	STAR_STAR_STAR_START;
}
```

Fonte/host.cpp

```
#include<Windows.h>

#Incluindo "constants.h"
#Incluindo "Trigerbot.H"
#Incluindo "PlayerGeometry.h"
#Incluindo "Menu.h"

Robótico* Robótico* Robótica acionada;
Geometria do jogador *Geometria do jogador;
Menu* Menu;

HMODULE Openglhandele=NULL;
void(__STDCal*GLDEPTHUNC) (unsigned int)=NULL;
DWORD OpenGl_Ret_Adress=0;

DWORD edi_value=0;
```

```
DWORD old_protect;

//Code cave, responsável por desativar os testes de profundidade do modelo
__declspec(naked)void opengl_codecave(){
	_asm{
		Pushad
	}

	if(menu->item_enabled[WALLHACK]){
		(*glDepthFunc)(0x207);
	}

	//Por fim, restaure o comando original e salte de volta
	_asm{
		Rebote
		mov esi, dword ptr ds:[esi+0xA18]
		jmp opengl_ret_address
	}
}

//Nossa Caverna de Código TrigerBot
__declspec(naked)void Trigerberbot_Codecave(){
	//Os blocos Asm permitem que você escreva montagens puras
	//Neste caso, usamos isso para chamar a função à qual estamos vinculados e
salvar todos os registros
	//Depois que fizemos a chamada, movemos seu valor de retorno em eax para uma
variável
	_asm{
		Chamar Triggerbot_ori_Call_Adress
		Pushad
		mov edi_value, eax
	}

	if(menu->item_enabled[TrigerBot]){
		Trigerbot->Experiência(edi_valor);
	}

	//Restaurar o registro e pular de volta para o CODE_ASM{

		Rebote
		jmp TrigerBot_or_Jump_Adress}

}
```

```
//Função assistente para impressão de texto
void Priment_Text (DWORD x, DWORD y, const char*text){
	Se (x>2400||x<0|y<0|y>1800){
		text=";
	}

	x+=200;

	_asm{
		mov ecx, texto
		Empurrar e soma
		Empurrar x
		Texto de chamada _Adress
		Adicionar esp, 8
	}
}

//Nossa caverna de código é responsável por imprimir o texto
__declspec(naked)void text_codecave(){
	Primeiro, recrie a função original à qual estamos vinculados, mas deixe o
texto em branco
	_asm{
		mov ecx, empty_text
		Texto de chamada _Adress
		Pushad
	}

	for(int i=0;i <MAX_ITEMS;i++){
		Priint_Text (50.250+(100*i), menu ->Item [i]); Priint_
		text (500, 250+ (100*i), menu ->get_stat(i));
	}
	 Priente_Text (10,250+ (100*menu->Cursor_Posição), menu->cursor);

	if(menu->ittem_Enabled[ESP]){
		//Em seguida, percorra todos os jogadores atuais no jogo em loop,
		por (int i=1;i<*playerGeometry->current_players;i++){
			Priente_Text (Playgeometry->x_values[i], Playgeometry->y
values[i], Playgeometry->Nomes[i]);
		}
	}

	//Restaurar o registro e retornar ao código original _asm{

		Rebote
		jmp ESP_Ret_Adress
```

```
      }
}

//Este thread contém todos os nossos códigos aimbot, ESP e OpenGL
void injectid_THREAD(){

      while(true){
            Se(Openglhandel==NULL){
                  Openglle=GetModuleHandle (L"opengl32.dll");
            }

   if(Open GlDHANDLE!=NULE & GlDHANDLE ==NULE){GlDHANDLE =(void(__StDHANDL*)(
                                    unsigned)
int) GetProcadodres (openGLHandle, "glDepthFunc");

                  //Então encontramos a posição do glDrawElements e o deslocamos
para um comando fácil de ganhar
                  unsigned char*OpenGl_Hook_Lokadeion=(unsigned
char*) getProcadernets ("
                  Opengl_hook_localização+=0x16;

                  //Para ganchos, removemos a proteção da memória no código que
queremos escrever
        //Em seguida, defina o primeiro código de operação para E9 ou salte
                  //Usando a fórmula para calcular a posição: New_Localização-
Originação+5
                  //Por fim, como a primeira instrução original tem um total de 6
bytes, o NOP emite os últimos bytes restantes
                  Projeto Virtualítico ((void*)Opengl_Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
                  *OpenGl_Hook_Localização=0xE9;
                  *(DWORD*)(Opengl_hook_localização+1)=
(DWORD)&opengl_codecave-((DWORD)OpenGL_Hook_Locação+5);
                  *(OpenGl_Hook_Localização+5)=0x90;

                  //Como o OpenGL é carregado dinamicamente, precisamos calcular
dinamicamente o endereço de retorno
                  opengl_ret_address=(DWORD) (OpenGl_Hook_Locação+0x6);}


            Menu->handle_input(); Player
            Geometry->update(); if(menu ->
            ittem_Enabled[AIMBOT]){
```

```
                  playerGeometry->set_player_view();
            }

            //Então nosso thread não funciona o tempo todo, fazemos com que ele
pague a execução por um milissegundo.
             //Isso permite que o processador agende outras tarefas.
            Sono (1);
      }
}

//Quando nosso DLL é carregado, criamos uma thread no processo para criar um gancho
. //Precisamos fazer isso porque nosso DLL pode ser carregado antes que o processo
carregue o OpenGL.
//Também criamos threads aimbot e ESP e vinculam a localização do TiggerBOT e do
texto impresso
BOOL WINAPI DllMain (Hinsttdl, DWORD fdwReason, LPVOID lpvreservid){

      unsigned char*Trigerbot_Hook_Location=(unsigned char*)0x0040AD9D;
      unsigned char*text_hook_localização=(unsigned char*)0x0040BE7E;

      Se(fdwReason==DL_Process_Attach){
            Tiggerbot=novo Tiggerbot();
           PlayerGeometry=Nova PlayerGeometry (0x509B74,0x50F4F8,
0x50F500);
            Menu = novo menu ();

            Reatethread (NULL, 0, (lpthead_star_routine) injected_thread, NULL
, 0, NULL);

            Projeto Virtualícola ((void*)Trigerbot_Hook_Locasion,5,
página_Express_Readwrite, &old_protect);
            *TrigerBot_Hook_Localização=0xE9;
            *(DWORD*)(trigerbot_hook_localização+1)=
(DWORD)&TrigerBot_Codecave-((DWORD)TrigerBot_Hook_Locação+5);

            Projeto Virtualítico ((void*)Text_Hook_Locatione, 5,
página_Express_Readwrite, &old_protect);
            *text_hook_localização=0xE9;
            *(DWORD*)(text_hook_localização+1)=(DWORD)&text_codificação
-((DWORD)text_hook_localização+5);
      }
      else if (fdwReason==DL_Process_Detach){
            delete Trigerbot;;;
            Excluir Player Geometry
            Excluir menus;
```

```
        }

        Retornar a verdade;
}
```

A.16 Wesnoth
Robô multijogador

Veja Capítulo 6.2.

Um cliente de amostra que será conectado ao servidor local Wesnoth 1.14.9 sob o nome de usuário FFFAAKKeee.

A maior parte do código é baseada no exemplo Winsock fornecido pela Microsoft: https://docs.microsoft.com/en-us/windows/win32/winsock/complete-client-code

```
#include<winsock2.h>
#include<ws2tcpp.h>
#Incluindo <stdio.h>

#Comentário de pragma (lib, "Ws2_32.lib")

#define DEFUL_BUFLEN 512

int main(int argc, char**argv){
        WSADA WSADA;
        Socket Conexão Socket = inválido_Socket;
        struct addrinfo*result=NULL,
                *ptr=vazio,
                Dicas;
        char recvbuf[DEFAULT_BUFLEN];
        int iResult
        int recvuflen=DEFULT_BUFLEN;

        //Solicitação de aperto de mão
        const unsigned char buff_handshake_p1[]={0x
                00,0x00,0x00,0x00
```

```
	};

	//Inclui versão 1.14.9
	const unsigned char buff_handshake_p2[]={
		0x00,0x00,0x00,0x2f, 0x1f, 0x8b, 0x0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0x2b, 0x2b, 0
		x2b, 0x2a, 0x8b, 0x00,0x00,0xxx0x0x0x0x0x0d2,5, c4,5, c4,5, c4,5, c4,5, c4,5, c4,5, c4,5, c4,5, c
		4,5, c4,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5,
		c,5, c,5, c,5, c,c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5,
		c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5,
		c,5, c,c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5, c,5,


	};

	//Inclui o nome de usuário FFFAAKKEEE
	const unsigned char Buff_Send_Nome[]={
		0x00, 0x00, 0x00, 0x3a, 0x1f, 0X08B, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
		0x8b, 0xce, 0xc9, 0x4f, 0X00008B, 0X08B, 0X08B, 0X0008B, 0X074, F75, 4, F76, 4
		, 4, A, 4, 4, 4, 4, 4, 4, 4, B, B, 4, 4, 4, 4, 4, 4, B, B, 4, 4, B, B, 4, B, 4, 4, B, B,
		4, 4, 4, B, 4, 4, 4, 4, 4, 4, B, 4, B, 4, B, B, 4, 4, 4, 4, 4, B, B, 4, 4, 4, 4, 4, 4, 4
		, 4, B, 4, 4, 4, B, B, 4, B, 4, 4, 4, 4, 4, B, B, B, 4, 4, B, 4, 4, B, B, B, B, 4, 4, 4,
		4, B, 4, 4, 4, 4, 4, 4, B, 4, 4, 4, 4, 4, B, 4, B, 5, 4, 4, 4, 4, 4, 4, B, 4, 4, 4, 4, 4
		, 4, 4, 4, 4, 4, B, 4, 5, 4, B, 4, 4, 5, B, 5, 5, 4, 4, B, 4, 5, 5, 4, B, 4, 5, 4, B, 4,


	};

	iResult = WSA Startup (2,2), &WSA Data);
	Se(iResult!=0){
		printf ("WSAStartup failed: %d\n", iResult);
		Retorno a 1;
	}

	Zeromory (hints, hints);
	hints.ai_family=af_int;
	hints.ai_socktype=SOCK_STREAM;
	hints.ai_protocol=IPPROTO_TCP;

	iResult=getaddrinfo ("127,0,0,1", "15000", &hints, &result); Se
	(iResult!=0){
		printf ("getaddrinfo falha: %d\n", iResult);
		wsaacleanup ();
		Retorno a 1;
	}

	ptr = resultado;

	Conect = socket (ptr->ai_family, ptr->ai_socktype, ptr-
>ai_protocol);
```

```
      Se(conectiveSocket==INVALID_SOCKET){
            printf ("Erro de soquete (): %ld\n", wsagetlasterrol());
            freeaddrinfo (resultados);
            wsaacleanup ();
            Retorno a 1;
      }

      Iresuult = Conect (Conect, ptr->ai_addr, (int)ptr->Ai_Addler); Se(
      iResult==SOCKET_ERROR){
            Closescocket (conectividade);
            Conect = Invalalid_Socket;
      }

      freeaddrinfo (resultados);

      Se(conective_Socket==INVALID_SOCKET){
            printf ("Não consegue se conectar ao servidor!\n");
            wsaacleanup ();
            Retorno a 1;
      }

      iResult=send(Conect, (const char*)buff_handshake_p1, (int)
sizeof(buff_handshake_p1), 0);
      printf ("Bytes enviados: %ld\n", iResult);

      Iresuult = recv (conect, recvbuff, recvbuff, 0); printf ("
      Bytes recebidos: %d\n", iResult);

      iResult=send(Conect, (const char*)buff_handshake_p2, (int)
sizeof(buff_handshake_p2), 0);
      printf ("Bytes enviados: %ld\n", iResult);

      Iresuult = recv (conect, recvbuff, recvbuff, 0); printf ("
      Bytes recebidos: %d\n", iResult);

      iResult=send(conexão, (conexão car*)buff_Send_Nome, (int)
sizeof(buff_Send_Nome), 0);
      printf ("Bytes enviados: %ld\n", iResult);

      De {
            Iresuult = recv (conect, recvbuff, recvbuff, 0); Se (
            iResult>0)
                  printf ("Bytes recebidos: %d\n", iResult);
            Caso contrário, se (iResult===0)
```

```
                printf("Conexão fechada\n");
            else
                printf ("recv falhou, erro %d\n", wsagetlasterrol()); } while
      (iResult>0);


      Closescocket (conectividade
      ); wsaacleanup ();

      Retornar a 0;
}
```

A.17 Wesnoth Chatbot

Veja Capítulo 6.4.

Um chatbot de exemplo que se conectará ao servidor local Wesnoth 1.14.9 usando o nome de usuário chatbot e responderá ao comando\wave.

A maior parte do código é baseada no exemplo Winsock fornecido pela Microsoft: https://docs.microsoft.com/en-us/windows/win32/winsock/complete-client-code

```
#Incluindo <stdio.h>
#include<winsock2.h>
#include<ws2tcpp.h>

#Comentário de pragma (lib, "Ws2_32.lib")

#include<zlib.h>

#define DEFUL_BUFLEN 512

void send_data (condens*data, tamanho _t len, SOCKET s){gzFile temp_
data = gzopen ("packet.gz", "wb");
gzwrite (temp_data, data, len);
gzclose (temp_data);
```

```
file*temp_filo=NULL;
fopen_s(&temp_filo, "packet.gz", "rb");

if(temp_filo){
size_t compress_len=0;
Buffer de caracteres sem símbolos [DEFAULT_BUFLEN]={0};
Comples_len=read (buffer, 1, tamp_filer); fclose (temp_filme);


unsigned char buff_packet[default_buffen]={0}; memcpy (buff_packet+3, &
compress_len, tamanho (compress_len)); memcpy (buff_packet+4, buffer,
compress_len);

int ireless=send(s, (const char*)buff_packet,compress_len+4,0); printf ("Bytes
enviados: %ld\n", iResult);
}
}

bool parse_data (unsigned char*buff,int buff_len){
Dados de caracteres sem símbolos [DEFAULT_BUFLEN]={0};
memcpy (dados, buff+4, buff_len-4);

file*temp_filo=NULL;
fopen_s(&temp_filo, "packet_recv.gz", "wb");

if(temp_filo){
fwrite (data, 1, sizeof (data), temp_file);
fclose (temp_filme);
}

gzFile temp_data_in=gzopenen ("packet_recv.gz", "rb");
unsigned char descompresse_Data[defaut_buflen]={0};
gzread (temp_data_in, decompresse_data, default_bufflen);
fwrite (decompresse_data, 1, default_bufflen, estoque);
gzclose(temp_data_in);

return Strengr(((const char*)data, (const char*)"\\\wave");}


int main(int argc, char**argv){
WSADA WSADA;
Socket conect Socket = inválido_Socket;
struct addrinfo*result=NULL, *ptr=NULL,
```

```
Dicas;
Caracteres não símbolos recvbuf[DEFAULT_BUFLEN];
int iResult
int recvuflen=DEFULT_BUFLEN;

//Solicitação de aperto de mão
const unsigned char buff_handshake_p1[]={
0x00,0x00,0x00,0x00
};

iResult = WSA Startup (2,2), &WSA Data);
Se(iResult!=0){
printf ("WSAStartup failed: %d\n", iResult);
Retorno a 1;
}

Zeromory (hints, hints);
hints.ai_family=af_int;
hints.ai_socktype=SOCK_STREAM;
hints.ai_protocol=IPPROTO_TCP;

iResult=getaddrinfo ("127,0,0,1", "15000", &hints, &result); Se(
iResult!=0){
printf ("getaddrinfo falha: %d\n", iResult);
wsaacleanup ();
Retorno a 1;
}

ptr = resultado;

Conect = socket (ptr->ai_family, ptr->ai_socktype, ptr-
>ai_protocol);

Se(conectiveSocket==INVALID_SOCKET){
printf ("Erro de soquete (): %ld\n", wsagetlasterrol());
freeaddrinfo (resultados);
wsaacleanup ();
Retorno a 1;
}

Iresuult = Conect (Conect, ptr->ai_addr, (int)ptr->Ai_Addler); Se(iResult==
SOCKET_ERROR){
Closescocket (conectividade); Conect =
Invalalid_Socket;}
```

```
freeaddrinfo (resultados);

Se(conective_Socket==INVALID_SOCKET){
printf ("Não consegue se conectar ao servidor!\n");
wsaacleanup ();
Retorno a 1;
}

iResult=send(Conect, (const char*)buff_handshake_p1, (int)sizeof(
buff_handshake_p1), 0);
printf ("Bytes enviados: %ld\n", iResult);

iRecort = Recort(Conect, (char*)Recort, Recort, 0);
printf ("Bytes recebidos: %d\n", iResult);

const unsigned char version[]="[version]\nversion=\"1.14.9\"\n[/version]";

send_data (versão, versão);

iRecort = Recort(Conect, (char*)Recort, Recort, 0);
printf ("Bytes recebidos: %d\n", iResult);

const unsigned char Nome[]="[login]\nusername="ChatBot\"\n[/login]"; send_
data (nome, sizeof (nome), conect);

const Unsigneeded char first_message[]="[message]\Nessage=\"ChatBot
Connect\"Nessage=\"Nessage=\"Nessage=\"ChatBot\"Nessage=\"ChatBot\"
Nessage"; send_data (first_message, sizeof (first_message), Conect);

De {
iRecort = Recort(Conect, (char*)Recort, Recort, 0); Se (iResult>0)

printf ("Bytes recebidos: %d\n", iResult);
Caso contrário, se (iResult===0)
printf("Conexão fechada\n");
else
printf ("recv falhou, erro %d\n", wsagetlasterrol());

Se(parse_data (recvbuf, iResult)){
const unsigned char message[]="[message]\Nessage=\"Olá! \"\nroom=\"lobby\"
\nsender=\\"ChatBot\"\n[/message]"; send_data (message, message (message),
Conectlock);}

} while(iResult>0);
```

```
Closescocket (conectividade);
wsaacleanup ();

Retorne a 0;}
```

A.18 Agente Wesnoth

Ver Capítulo 6.5.

O agente de exemplo do Wesnoth 1.14.9 permite a interceptação e modificação do tráfego do cliente do jogo Wesnoth para o servidor do Wesnoth. Neste caso, sempre que o agente vir uma mensagem de bate-papo\wave, ele enviará uma mensagem de bate-papo adicional dizendo olá!

A maior parte do código é baseada nos exemplos de Winsock fornecidos pela Microsoft: https://docs.microsoft com/en-us/win32/Winsock/complete-client-code e https://docs.microsoft.com/en-us/windows/win32/winsock/complete-server-code

```
#include<winsock2.h>
#include<ws2tcpp.h>
#Incluindo <stdio.h>

#Comentário de pragma (lib, "Ws2_32.lib")

#define DEFUL_BUFLEN 512
#define DEFULT_Porto "27015"

#include<zlib.h>

void send_data (condens*data, tamanho _t len, SOCKET s){gzFile temp_
data = gzopen ("packet.gz", "wb");
gzwrite (temp_data, data, len);
gzclose (temp_data);

file*temp_filo=NULL;
fopen_s(&temp_filo, "packet.gz", "rb");

if(temp_filo){
size_t compress_len=0;
Buffer de caracteres sem símbolos [DEFAULT_BUFLEN]={0};
```

```
Comples_len=read (buffer, 1, tamp_filer); fclose (temp_filme);


unsigned char buff_packet[default_buffen]={0}; memcpy (buff_packet+3, &
compress_len, tamanho (compress_len)); memcpy (buff_packet+4, buffer,
compress_len);

int iResult=send(s, (const char*)buff_packet,compress_len+4,0); printf ("Bytes
enviados: %ld\n", iResult);
}
}

bool parse_data (unsigned char*buff,int buff_len){
Dados de caracteres sem símbolos [DEFAULT_BUFLEN]={0};

memcpy (dados, buff+4, buff_len-4);

file*temp_filo=NULL;
fopen_s(&temp_filo, "packet_recv.gz", "wb");

if(temp_filo){
fwrite (data, 1, sizeof (data), temp_file);
fclose (temp_filme);
}

gzFile temp_data_in=gzopenen ("packet_recv.gz", "rb");
unsigned char descompresse_Data[defaut_buflen]={0};
gzread (temp_data_in, decompresse_data, default_bufflen);
fwrite (decompresse_data, 1, default_bufflen, estoque);
gzclose(temp_data_in);

return Strengr(((const char*)data, (const char*)"\\\wave");}


int main (void)
{
WSADA WSADA;
int iResult

Lista de sockets sockets = invalid _sockets; Socket cliente Socket = inválido _
Socket; SOCKET ServerSocket=INVALID_SOCKET;


struct addrinfo*result=NULL, prompt;
```

```
int iSendResult;
Caracteres não símbolos recvbuf[DEFAULT_BUFLEN];
int recvuflen=DEFULT_BUFLEN;

Tempo limite do DWORD = 1000;

//Socket do cliente
iResult = WSA Startup (2,2), &WSA Data);

Zeromory (hints, hints);
hints.ai_family=af_int;
hints.ai_socktype=SOCK_STREAM;
hints.ai_protocol=IPPROTO_TCP;
hints.ai_flags=AI_PASSIVE;

iResult=getaddrinfo (NULL, DEFAULT_PORT, &hints, &RELT); ListenSocket=ListenSocket
(ai_family, ListenSocket, ListenSocket = ListenSocket (ListenSocket, ListenSocket)
>ai_protocol);
Iresolt=bind(ListenSocket, resolt->ai_addr, (int) resolt->ai_addren);
freeaddrinfo (resultados);

Iresolt=listen (ListenSocket, SOMAXCONN);
ClientSocket = acept (Listentsocket, NULL, NULL);
Closseocket (ListenSocket);

//Socket do servidor
Zeromory (hints, hints);
hints.ai_family=af_int;
hints.ai_socktype=SOCK_STREAM;
hints.ai_protocol=IPPROTO_TCP;

iResult=getaddrinfo ("127,0,0,1", "15000", &hints, &result); Verserocket=
socket (resulit->ai_family, resulit->ai_SOCTYPE, resulit->ai_protocol);

iResult=Conect(Serversoft, result->ai_addr, (int)result-
>ai_addrlen);
freeaddrinfo (resultados);

SETSOCKET (SETSOCKET, SO_RCVTIMEO, (char*)&timeout, SETEOUT (timeout);


De {
Iresuult = recv(ClientSocket, (char*)Recvbuff, Recvbufflen, 0); Sono (100);
```

```
Se(iResult>0){
printf ("Bytes recebidos: %d\n", iResult); Se(parse_
data (recvbuf, iResult)){
const unsigned char message[]="[message]\Nessage=\"Olá! \"\nroom=\"lobby\"\
nsender=\\"ChatBot\"\n[/message]"; send_data (message, message (message),
ServerSocket); Sono (100);

}

iSendResult = send (server socket, (char*)recvbuf, isult, 0); Sono (100);

printf ("Bytes enviados: %d\n", iSendResult); iResult=recv(ServerSocket, (
char*)Recvbuff, Recvbufflen, 0); Sono (100);

if(iresult_EROR){iresult_EROR) {iresult_SEROR = iresult_SEROR (CLINTSEROR, (
CLINTSEROR, Iresult_SEROR);

}
}
Caso contrário, se (iResult===0)
printf ("Conexão está fechada...\n");
else
printf ("recv falhou, erro %d\n", wsagetlasterrol()); } while(iReSun>0||
WSAGETLETEROR()==WSAETEROUT); Iresuult=shutdown (clínica, SD_SEND);




closesocket (ClientSocket); closesocket (socket do servidor); wsaacleanup ();



Retorne a 0;}
```

Injetor de DLL A.19

Ver Capítulo 7.1.

O injetor de DLL que carrega o DLL especificado no Urban Terrol.

Para carregar bibliotecas estáticas e dinâmicas, os executáveis Windows podem ser usados usando a função API Loadlibrearya. Esta função aceita um parâmetro que é o caminho completo para a biblioteca a ser carregada.

```
HMODULE LoadLibraryA (
LPCSTR LPLIBLEAME);
```

Se chamarmos Laoadlibrearya no código do injetor, o DLL será carregado na memória do injetor. Em vez disso, queremos que nossos injetores forcem o jogo a chamar o loadlibraya . Para isso, criaremos um novo thread no jogo usando a API CereerEMOTETHRead. O thread então executa o loadlibrearya enquanto o jogo está em execução.

Entretanto, como os threads estão executando na memória do jogo, o Ladlibrearya não conseguirá encontrar o caminho para o DLL especificado no injetor. Para resolver esse problema, precisamos gravar o caminho do DLL na memória do jogo. Para garantir que não danifiquemos nenhuma outra memória, também precisamos usar o VirtuAllocex para alocar memória adicional no jogo.

```
#include<windows.h>
#include<tlhelp32.h>

//O caminho completo do DLL a ser injetado.
const char*DL_Path="C:\\Users\\\IEUser\\\source\\repos\WallHaack\Debug\
WallHaack\WallHaack\dl";

int main(int argc, char**argv){
	Instantâneo do cabo = 0;
	processEntry32 pe 32={0};

	Código de saída DWORD = 0;

	pe32. dwSize = dwSize (PROCESSENTRY32);
```

```
      //O código do snapshot é Microsoft em

      //https://docs.micropsoftcom/en-us/win32/toolhelp/Taoking-A-Snapshot-
and-Viewing-Processos
      snapshot=Creatotolhelp32Snapshot (th32cs_Snapshot, 0);
      process 32 first (instantâneo, &P32);

      De {
            //Queremos apenas operar o processo de terror urbano se (
            wcscp(pe32.szExeFile, L"Quake3-UrT.exe")==0){
                  Primeiro, precisamos obter um cabo de processo para as seguintes
chamadas
                  Processo de cabo = Processo aberto (Process_All_Acesso, verdade,
PE32.Th32Procesid);

                  //Então não destruiremos nenhuma memória e alocaremos memória
adicional para salvar nosso caminho DLL
                  void*lpbazadres = Virtuallock (dll_path)+1, MEM_
comando, página_readwrite);

                  //Escreva nosso caminho DLL na memória que acabamos de alocar
dentro do jogo
                  WriteProcess (process, lpbase Address, DL_Path, strern(DL_
Path)+1, NULL);

                  //Crie um thread remoto no jogo para executar loadlibriary

                  //Para esta chamada de loadlibrigary, passaremos o caminho
completo do DLL que escrevemos no jogo
                  HMODULE KENEL32BASE=GETMODULE (L"KENEL32.DLL"); Thread do
                  cabo = Createterhotethread (process, NULL, 0,
(lpthead_Start_Routine) GetProcadernets (kernel32base, "loadlibrearya"),
lptheadred, 0, NULL);

                  Para garantir que nosso DLL seja injetado, podemos usar as duas
chamadas a seguir para bloquear a execução do programa
                  WaitForsingleock (thread, ilimitado);
                  GetExitCodeThread(thread, &exitCode) ;

                  //Por fim, libere a memória e limpe o manual do processo
                  VirtualFreeX (process, lpbase Address, 0, MEM_Reass); Feche o
                  cabo (thread);
                  Feche o cabo (processo
                  ); descanso;
```

```
            }
      }while(Process32Next(instantâneo, &pe32
      )); Retornar a 0;

}
```

Scanner de padrão A20

Ver Capítulo 7.2.

Um scanner de padrão pesquisará bytes 0x2942 04 no processo Wesnoth em execução . Esses bytes são os códigos de operação de subcomandos, que são responsáveis por deduzir moedas de ouro dos jogadores ao recrutar uma unidade.

O scanner funciona usando o Creatotolhelp32Snapshot para encontrar o processo Wesnoth e o módulo Wesnoth primário. Uma vez localizado, um buffer é criado e a memória do módulo é lida nele. A memória do módulo contém principalmente códigos de operação para instruções. Uma vez carregado, passamos por todos os bytes no buffer e pesquisamos nossos padrões. Uma vez encontrado, imprimimos e impressão offset.

```
#include<windows.h>
#include<tlhelp32.h>
#Incluindo <stdio.h>

//O modo de código de operação que queremos escanear dentro do processo
Bytes de caracteres não símbolos []={0x29,0x42,0x04}

int main(int argc, char**argv){
      Mancha de frase process_snapshot=0;
      module_snapshot=0;
      processEntry32 pe 32={0};
      Entrada do módulo 32 ME32;

      Código de saída DWORD = 0;

      pe32. dwSize = dwSize (PROCESSENTRY32);
      me32 2. dwSize = dwSize (moduleentry 32);

      //O código do snapshot é Microsoft em
```

```
      //https://docs.micropsoftcom/en-us/win32/toolhelp/Taoking-A-
Snapshot-and-Viewing-Processos
      process_snapshot=Creatotolhelp32snapshot(th32cs_snapshot, 0);
      Process32First(process_snapshot, &pe32);

      De {
            //Se(wcscmp(pe32 2. szExeFile, L"wesnoth.exe") ==0){
            escaneie somente os padrões dentro do processo Wesnoth
                  module_snapshot=
Creatoteolhelp32Snapshot(TH32CS_SNAPMODULE, pe32.th32ProcessID);

                  //Recuperar um cabo de processo para que possamos ler o cabo do
Memór               jogo process=OpenProcess(PROCESS_ALL_ACCESS, verdade,
ia
PE32.Th32Procesid);

                  Module32First(module_snapshot, &me32);
                  De {
                        //Wesnoth consiste em muitos módulos. Para nosso
exemplo, queremos apenas escanear o código do módulo principal executável
                              if(wcscmp(me32.szmodul, L"wesnoth.exe") ==0){//
                                    Devido ao tamanho do código, dinamicamente
Crie um buffer após determinar o tamanho
                              unsigned char*buffer=(unsigned)
char*)Caloc(1 me32. Modbassize);
                              DWORD bytes_read=0;

                              //Leia todo o bloco de código em nosso buffer
                              ReadProcessMemory(process,
(void*)me32.modbasidr, buffer, me32.modbasidar, &bytes_read);

                              //Para cada byte no código do jogo, verifique se o
padrão do byte começa com esse byte
                              Para(unsigned int i=0; i<me32 2. Modbassize-
sizeof(bytes); i++){
                                    for(int j=0;j<sizeof;j++){//Se sim,
                                          continue verificando se todos
Correspondência de bytes. Caso contrário, saia do loop
                                          Se(bytes[j]!=Buffer[i+j]){
                                                interromper;
                                          }

                                          //Se estivermos no final do loop,
todos os bytes devem corresponder
                                          Se(j+1==sizeof(bytes)){
```

```
                                                    printf("%x\n", i+
(DWORD)me32.modbasidr);
                                                }
                                            }
                                        }

                                        Livre (tampão);
                                        descanso;
                                    }

                            } while(module_snapshot,&me32);

                            Feche o cabo (processo
                            ); descanso;
                        }
                }while(process_snapshot,&pe32); Retornar a 0;


}
```

Memória A21 scanner

Ver Capítulo 7.3.

O scanner de memória da Wesnoth permite que você pesquise, filtre e edite memória durante o processo. Este código pode ser adaptado a qualquer objetivo e foi projetado para mostrar como ferramentas como mecanismos de trapaça funcionam.

O scanner tem três operações principais:

- Pesquisar
- Filtro
- Escrever

A operação de pesquisa escaneia toda a memória de 0x00000000 a 0x7fffffff e usa ReadProcessMemory para determinar se o endereço salva um valor. Porque

ReadProcessMemory falha Se o processo não tiver acesso a um endereço, a memória será escaneada por bloco. Qualquer valor correspondente é salvo em res.txt.

As operações do filtro iteram todos os endereços em res.txt para determinar se eles correspondem aos valores fornecidos. Em caso afirmativo, salve-os em Rees_fil.txt. Por fim, res_fil.txt foi copiado para res.txt.

A operação de gravação usa WriteProcessMemory para escrever o valor passado em todos os endereços em res.txt

Creatoteolhelp32Snapshot é usado para pesquisar processos Wesnoth, e OpenProcess é usado para pesquisar manipuladores.

```
#include<windows.h>
#include<tlhelp32.h>
#Incluindo <stdio.h>

#Definição de tamanho 0x00000808

//A função de pesquisa escaneia toda a memória de 0x00000000 a 0x7fffffff para
encontrar valores transferidos
void search (const HANDLE process, const int passed_val){
      file*temp_filo=NULL;
      fopen_s(&temp_filo, "res.txt", "w");

      unsigned char*buffer=(unsigned char*)Calc (1, tamanho);
      DWORD bytes_read=0;

      for (DWORD i=0x000000; i<0x7fffffff;i+=tamanho) {process, (vod*)i, buffer
            , tamanho, &bytes_read);

            for (int j=0; j<tamanho-4; j+=4) {DWORD
                  val=0;
                  MEMCY (&val, &buffer[j], 4);
                  Se(val==passed_val){
                        fprintf(temp_filo, "%x\n", i+j);}

            }
      }

      fclose (temp_filme
      ); Livre (tampão);

}
```

```
A função //filter obtém uma lista de endereços em res.txt e verifica se //elas
correspondem aos valores fornecidos. Se sim, escreva-os em res_fi.txt//Após a
aprovação inicial, o filtro escreve todos os endereços em res_fi.txt para res.txt

void filter (const HANDLE process, const int passed_val){
      file*temp_filo=NULL;
      file*temp_file_file_filter=NULL;
      fopen_s(&temp_filo, "res.txt", "r");
      fopen_s(&temp_file_filtro, "res_file.txt", "w");

      Endereço DWORD = 0;
      while(fscanf_s(temp_file, "%x\n",&adress)! =EOF){
            DWORD val=0;
            DWORD bytes_read=0;

            ReadProcessMemory (processo, (void*) endereço, &val, 4, &bytes_read
            ); Se(val==passed_val){
                  fprintf(temp_file_filter, "%x\n", endereço);
            }
      }

      fclose (temp_filme);
      fclose (temp_file_filter);

      fopen_s(&temp_filo, "res.txt", "w");
      fopen_s(&temp_file_filtro, "res_file.txt", "r"); while(fscanf_
      s(temp_file_filter, "%x\n",&adres)! =EOF){
            fprintf (temp_fil, "%x\n", endereço);
      }

      fclose (temp_filme);
      fclose (temp_file_filter);

      remove ("res_fil.txt");
}

//A função write grava um valor para cada endereço em res.txt: void
write (const Hand process, const Hand passed_val){
      file*temp_filo=NULL;
      fopen_s(&temp_filo, "res.txt", "r");

      Endereço DWORD = 0;
      while(fscanf_s(temp_file, "%x\n", &adss)! =EOF){DWORD
            bytes_written=0;
```

```
            WriteProcess(process, (void*)Adresses, &passed_val, 4,
&BYTE_WRITTEN);
      }

      fclose(temp_filme);
}

//A função principal é recuperar o cabo do processo de Wesnoth, analisar os
parâmetros do programa e passar
//Executar a operação correta
int main(int argc, char**argv){
      Maneira de frase process_snapshot=0;
      processEntry32 pe 32={0};

      pe32. dwSize = dwSize (PROCESSENTRY32);


      //O código do snapshot é Microsoft em

      //https://docs.micropsoftcom/en-us/win32/toolhelp/Taoking-A-Snapshot-
and-Viewing-Processos
      process_snapshot=Creatotolhelp32snapshot (th32cs_snapshot, 0); Process
      32First(process_snapshot, &pe32);

      De {
            //Recuperar apenas o cabo de processo do Wesnoth
            if(wcscmp(pe32. szExeFile, L"wesnoth.exe") ==0){
                  //Recuperar um cabo de processo para que possamos ler e escrever
a memória do jogo
                  Processo de mangue = Processo aberto (Process_All_Accesso
, verdade, pe32.th32Processo);

                  //Converter o segundo parâmetro para um valor semelhante
                  ao DWORD, char*p;
                  Valor comprimento = strtol (argv[2],&p, 10);

                  //Com base no primeiro parâmetro, a execução será transferida
para a operação de pesquisa, filtragem ou gravação
        Se(strcp(argv[1], "pesquisar")==0){pesquisar (processe, valor);

                  }
                  else if(strcp(argv[1], "filtro") ==0){
                        filtro (profil, valor);
                  }
                     else if(strcp(argv[1], "write")==0){
```

```
                        Escrever(processo, valor);
                }

                //Desligue o cabo do processo
                Closehandel(process);
                descanso;
        }
    }while(process_snapshot, &pe32); Retornar a 0;


}
```

A.22 Inversor

Ver Capítulo 7.4.

Um programa de desmontagem limitado que busca o processo Wesnoth em execução e então desmonta 0x50 bytes a partir de 0x7cd91. Essas instruções são responsáveis por deduzir moedas de ouro de um jogador ao recrutar uma unidade.

O programa de desmontagem funciona usando o Creatotolhelp32Snapshot para encontrar o processo Wesnoth e o módulo Wesnoth primário. Uma vez encontrado, um buffer é criado e a memória do módulo é lida nele. A memória do módulo contém principalmente códigos de operação para instruções. Uma vez carregados, nós loop todos os bytes no buffer e os desmontamos com base nas referências fornecidas pela Intel aqui.

```
#include<windows.h>
#include<tlhelp32.h>
#Incluindo <stdio.h>

#Definir endereço inicial 0x7cd91

//8 valores de operação possíveis
const char modrm_value[8][4]={
      "eax", "ecx", "ebx", "esp",
      "ebp", "esi",
```

```
      "Eddie"
};

//As tabelas 2-2 do documento de referência descrevem como recuperar operandos a
partir de valores ModR/M
int Decode_Oper (unsigned char*buffer, int location){if(buffer[
      location]>=0xC0&buffer[location]<=0xFF){
            Priintf ("%s, %s", modrm_valor[buffer[locação]%8],
modrm_valor[(buffer[locação]>3)%8]);
            Retorno a 1;
      }
      Caso contrário, se (buffer [Localização]>=0x80&buffer [Localização]<=0xBF){
            Deslocamento DWORD=buffer [Localização +1]|(buffer [Localização +2]
 <<<8)|(Buffer [posição + 3]<<<16) |(Buffer [posição + 4]<<24); Priintf([%s+%s], %s
                                                  ", modrm_value[buffer[Locação]%8],
Deslocamento, modrm_valor[(buffer[localização]>>3)%8]);
            Retorno a 5;
      }
      Esse if (buffer [Cocation]>=0x40&&buffer [Cocation]<=0x7F){printf
            ("[%s+%x], %s", modrm_value [Cocation]%8],
Buffer [Localização+1], modrm_value [(Buffer [Localização]>>3)%8]);
            Retorno a 2;
      }

      Retorno a 1;
}

int main(int argc, char**argv){
      Maneira de frase process_snapshot=0;
      module_snapshot=0;
      processEntry32 pe 32={0};
      Entrada do módulo 32 ME32;

      Código de saída DWORD = 0;

      pe32. dwSize = dwSize (PROCESSENTRY32);
      me32 2. dwSize = dwSize (Moduleentry 32);

      //O código do snapshot é Microsoft em

      //https://docs.micropsoftcom/en-us/win32/toolhelp/Taoking-A-Snapshot-
and-Viewing-Processos
      process_snapshot=Creatotolhelp32snapshot (th32cs_snapshot, 0); Process32
      First(process_snapshot, &pe32);
```

```
      De {
            //Dessinstalar apenas o processo Wesnoth
            if(wcscmp(pe32. szExeFile, L"wesnoth.exe") ==0){
                  module_snapshot=
Creatoteolhelp32Snapshot (TH32CS_SNAPMODULE, pe32.th32ProcessID);

                  //Recuperar um cabo de processo para que possamos ler a memória
do jogo
                  Processo de cabo = Processo aberto (Process_All_Acesso, verdade,
PE32.Th32Procesid);

                  Module32First (module_snapshot, &me32);
                  De {
                        //Wesnoth consiste em muitos módulos. Para nosso exemplo,
queremos apenas escanear o código do módulo principal executável
                        if(wcscmp(me32.szmodul, L"wesnoth.exe") ==0){
                              //Devido ao tamanho do código, um buffer é criado
dinamicamente após determinar o tamanho
                              unsigned char*buffer=(unsigned)
char*)Caloc (1 me32. Modbassize);
                              DWORD bytes_read=0;

                              //Leia todo o bloco de código em nosso buffer
                              ReadProcessMemory (process,
(void*)me32.modbasidr, buffer, me32.modbasidar, &bytes_read);

                              DWORD loc = 0;
                              unsigned int i=Start_Adress-
(DWORD)me32.modbase addr;

                              //Tente desmontar cada byte no código do jogo

                              while(i<Start_Adress+0x50-(DWORD)
me32.modbasidr){
                                    printf("%x:\t", i+
(DWORD)me32.modbasidr);
                                    Interruptor (buffer[i]){
                                    Caso 0x1:
                                          printf ("adicionar");
                                          i++;
                                          i+=decode_Opend (buffer, i);
                                          descanso;
                                    Caso 0x29:
                                          printf ("SUB");
                                          i++;
```

```
                                        i+=decode_Opend(buffer,i);
                                        descanso;
                                  Caso 0x74:
                                        printf("JE");
                                        printf("%x",i+
(DWORD)me32.modbase addr+2+buffer[i+1]);
                                        i+=2;
                                        descanso;
                                  Caso 0x80:
                                        printf("CMP");
                                        i++;
                                        i+=decode_Opend(buffer,i);
                                        descanso;
                                  Caso 0x8D:
                                        printf("LEA");
                                        i++;
                                        i+=decode_Opend(buffer,i);
                                        descanso;
                                  Caso 0x8B:
                                  Caso 0x89:
                                        printf("MOV");
                                        i++;
                                        i+=decode_Opend(buffer,i);
                                        descanso;
                                  Caso 0xE8:
                                        printf("chamada");
                                        i++;
                                        loc=Buffer[i]|(Buffer[i+1]<8)|(
Buffer[i+2]<<16)|(Buffer[i+3]<<24);
                                        printf("%x",loc+(i+(
DWORD)me32.modbasidr)+4);
                                        i+=4;
                                        descanso;
                                  Padrão padrão:
                                        printf("%x",buffer[i]);
                                        i++;
                                        Quebr
                                        ar;}

                                        printf("\n
                              ");}

                              Livre(tampão
                              );descanso;
                        }
```

```
                                    } while(module_snapshot,&me32);

                    Feche o cabo (processo);
                    descanso;
              }
       }while(process_snapshot,&pe32); Retornar a 0;


}
```

Depurador A23

Ver Capítulo 7.5.

Um exemplo de um depurador Windows que será anexado ao processo ASSULT Cube 1.2.0.2 em execução, alterando instruções específicas para instruções int 3 (0xCC) e, em seguida, restaurando as instruções originais quando um ponto de interrupção for encontrado. As instruções modificadas são executadas somente quando o player é acionado, permitindo-nos verificar se o depurador está funcionando conforme o esperado.

```
#include<windows.h>
#include<tlhelp32.h>
#Incluindo <stdio.h>

int main(int argc, char**argv){
Mancha de frase process_snapshot=NULL
; O cabo da frase é thread_handle=NULL;
Mancha de frase Process_Handlle=NULL;

processEntry32 pe 32={0};

DWORD pid
DWORD Continuation = DBG_Continuation;
DWORD BYTE_WRTEN=0;

Instrução de byte _break=0xcc;
Instrução de byte_normal=0x8b; DEBUG_
EVENT debugEvent={0};
```

```
Contexto contexto ={0};

bool first_break_has_occurred=false;

pe32. dwSize = dwSize (PROCESTRY32);

//Percorra todos os processos ativos e encontre o processo de cubo de ataque process_
snapshot=Creatotolhelp32Snapshot (th32cs_Snapshot, 0); Process32First(process_
snapshot, &pe32);

De {
if(wcscmp(pe32. szExeFile, L"ac_client.exe")==0){
//Salve o pid e escreva o comando int 3 para 0x0046366C pid = pe32.th32PROCSID;


process_handle=OpenProcess (PROCESS_ALL_ACCESS, verdade,
PE32.Th32Procesid);
WriteProcessMemory (process_handle, (void*) 0x0046366C,
&instruction_break, 1, &BYTES_WRTESTEN);
}
}while (process_snapshot, &pe32);

//Adicione um depurador e entre no ciclo principal de depuração
Processo de atividade de depuração (pid);

for(;;){
Continuestados = dbg_Continuestados;

Se (! debugEvent (&debugEvent, INFINITE))
Retornar a 0;

Interruptor (Debugevent.dwdebdeugeventcode){
Caso Excepcion_Debug_Event:
Interruptor (Debugevente.u.Exception Record.Exception Record.Exception Record.
Exception Code){
case EXCEPTION_BREAKPOINT:
printf ("ponto de ruptura acerto");

//Nosso código de ponto de interrupção principal
//Isso será atingido primeiro ao adicionar, então ignoramos essa condição pela
primeira vez.
if(first_break_has_occurred){
//Se interrompermos, abra o cabo do thread que acionou o evento e restaure o eip para a
instrução anterior
```

```
thread_handle=OpenThread (Thread_All_acesso, verdade, debugevent.dwhread);

Se (thread_handle!=NULL){
context.ContextFlags=CONTEXT_ALL; GettheadContext (thread_handle, &context);
Contexto. Eip-;



context (thread_handle, &context); CloseHandle (thread_handle);


//Em seguida, escreva de volta o comando mov anterior para que nosso ponto de
interrupção não seja acionado novamente
WriteProcess_Handle, (void*) 0x0046366C, &instruction_normal, 1, &BYTES_
WRTESTEST);
}
}

first_break_has_occurred=verdade;
Continuestados = dbg_Continuestados;
descanso;
Padrão padrão:
Continuestato=DBG_EXCEPTION_NOT_HANDLED; descanso;

}
descanso;
Padrão padrão:
Continuestato=DBG_EXCEPTION_NOT_HANDLED;
descanso;
}

ContinueeBogeVent (debugEvent. dwProcessId, debugEvent. dwProcessId,
ContinueeStat);
}

Caminho fechado (process_handle);
Retornar a 0;

}
```

Gravador de chamadas A.24

Ver Capítulo 7.6.

Um exemplo de um depurador Windows modificado que será anexado ao processo Wesnoth em execução, todas as instruções de chamada serão encontradas e alteradas para instruções int 3. Quando um ponto de interrupção é encontrado, a localização é registrada e o comando é restaurado . Em seguida, após a execução da instrução, a instrução int 3 será reescrita para esse local.

```
#include<windows.h>
#include<tlhelp32.h>
#Incluindo <stdio.h>
#Incluindo <pspapi.H>

#define Read_página_size 4096

int main(int argc, char**argv){
Mancha de frase process_snapshot=NULL;
O cabo da frase é thread_handle=NULL;
Mancha de frase Process_Handlle=NULL;

processEntry32 pe 32={0};

DWORD pid
DWORD Continuation = DBG_Continuation; DWORD
BYTE_WRTEN=0;

Instrução de byte _break=0xcc;
Byte Instruction_Chamada=0xe8;

DEBUG_EVENT debugEvent={0};

Contexto contexto ={0};

bool first_break_has_occurred=false;

Módulo H Módulo [128]={0}; MODULEINFO
module_info={0};

DWORD bytes_read=0;
Deslocamento DWORD = 0;
DWORD Call_Localização=0;
```

```
DWORD Call_Locate_BYTES_Read=0;
DWORD Last_Call_Localização=0;

Instrução de caracteres sem símbolos [read_página_tamanho]={0};

int breakpoints_set=0;

pe32. dwSize = dwSize (PROCESSENTRY32);

//Viaje por todos os processos ativos e encontre o Processo Wesnoth process_
snapshot=Creatotolp32Snapshot (th32cs_Snapshot, 0); Process32First(process_
snapshot, &pe32);

De {
if(wcscmp(pe32. szExeFile, L"wesnoth.exe") ==0){
//Salve o pid e abra o cabo do processo
pid=pe32.th32PROCSID;

process_handle=OpenProcess (PROCESS_ALL_ACCESS, verdade,
PE32.Th32Procesid);
}
}while(process_snapshot,&pe32);

//Adicione um depurador e entre no ciclo principal de depuração
Processo de atividade de depuração (pid);

for(;;){
Continuestados = dbg_Continuestados;

Se (! debugEvent (&debugEvent, INFINITE))
Retornar a 0;

Interruptor (Debugevent.dwdebdeugeventcode){
Caso Excepcion_Debug_Event:
Interruptor (Debugevente.u.Exception Record.Exception Record.Exception Code
){
case EXCEPTION_BREAKPOINT:
//Substitua todas as chamadas pelo ponto de interrupção do anexo inicial

if(! first_break_has_occurred){thread_handle=OpenThread (thread_al_
acesso, verdade, debugevente.dwhread);


printf ("Ponto de interrupção adicional\n");
```

```
Neste código, registraremos apenas todas as chamadas no módulo principal do jogo,
não DLL
//Para localizar o espaço de endereço deste módulo, pesquise todos os módulos e
obtenha o primeiro
//Espaço de endereço do módulo
Mobilizadores ENUMPOS (Process_Handles, Mobilizadores,
Size (move), &bytes_read);
Formação de GetModule_Handle, modules[0],
&module_info, sizeof (module_info);
//Em seguida, atravesse cada parte da memória em loop para encontrar o código de
operação chamado (0xe8)
Para (DWORD i=0;i<module_info.sizeofige;i+=
Read_página_tamanho){
//Se as permissões de memória da página estiverem incorretas, o ReadProcessMemory
falhará
//Para evitar que uma única falha pule toda a memória, leia uma página de
memória de uma só vez (process_handle, (LPVOID)

((DWORD)Module_info.lpbase ofdll+i), &instraccions, read_página_tamanho,
&bytes_read);
Para (DWORD c=0;c<bytes_read;c++) {//Se detectarmos 0xe8, determinamos se é uma
instrução de chamada

Primeiro, lemos os quatro bytes após 0xe8

//Então usamos esses bytes para calcular a localização da chamada

//Se esta localização estiver fora do espaço de endereço do módulo principal,
ignoramos o código de operação if(instractons[c]==instractons){offset=(DWORD)
move_info.lpbase ofdl+i+c;


ReadProcess_Handle, (LPVOID) (offset+1), &Call_Localiza,4.&Call_Localiza_
Localiza_Localiza_Localiza_Localiza_Localiza_Localiza_); Call_Localização+
=Oferta+5; Se (call_localização<

(DWORD)Module_info.lpbase ofdl|call_localização
>(DWORD)module_info.lpBaseOfDll+module_info.sizeofige)Continuar;


//Se a localização da chamada for válida, uma instrução de interrupção será escrita
naquele endereço (0xcc)
//Neste caso, 0x0040e3d8 e 0x0040e3ea são dois endereços comumente usados que contêm
código de baixo nível
```

```
//Para evitar que eles bloqueiem os registros, não registramos esses locais

//Além disso, ter milhares de pontos de interrupção pode fazer com que o programa em
execução caia. //Portanto, limitamos o número de pontos de interrupção a 2.000.


Se (oferta! =0x0040e3d8&&oferta! =
0x0040e3ea&&&BreakPotencia_set<2000){process_handle (void*) Offsette, &
Instrucia_Break,1,&Break_WRITTEN); process_handle (LPVOID) Offsette, 1);



breakpoints_set++;}

}
}
}

printf ("completar pontos de interrupção adicionais\n");}

Caso contrário{
//Se interrompermos, abra o cabo do thread que acionou o evento e restaure o eip à
instrução anterior. //Em seguida, definimos o modo de etapa única para que possamos
restaurar o ponto de interrupção.

//Depois disso, escreveremos a instrução de chamada e continuaremos a executar o
programa
thread_handle=OpenThread (Thread_All_acesso, verdade, debugevent.dwhread);

if(thread_handle!=NULL){Context.Contextflash=CONTEXT_ALL; Getthead
_Contextflash (thread_handle, &context);


Contexto. Eip-;
Contexto. EFlags|=0x100;

context (thread_handle, &context); CloseHandle (thread_handle);


WriteProcess_Handle, (void*)Context_eIP, &instruction_call, 1,
&BYTES_WRTEN); FlushinstructionCase(process_handle, (LPVOID)
context.Eip, 1);


Last_Call_Localização=context.Eip;
```

```
}
}

first_break_has_occurred=verdade;
Continuestados = dbg_Continuestados;
descanso;
Estudo de caso Excepcion_Single_Passo:
//Este código será executado depois que entrarmos no modo de etapa única na
declaração de ponto de interrupção acima
//Executar uma única instrução no modo de passo e acionar este evento de depuração

//Portanto, após executar a chamada interrompida acima, restauramos a instrução
break para que nosso ponto de interrupção não seja//acionado apenas uma vez

thread_handle=OpenThread (Thread_All_acesso, verdade, debugevent.dwhread);

Se (thread_handle!=NULL){
Context.Contextflash=CONTEXT_ALL; GettheadContext (thread_
handle, &context); CloseHandle (thread_handle);


WriteProcessMemory (process_handle,
(void*)Last_Call_Localização, &instruction_break, 1, &BYTES_WRTEN);
Flushinstruction Caque (process_handle,
(LPVOID)Last_Call_Locatione, 1);

printf ("0x%08x: chamada 0x%08x\n", last_call_localiza,context.Eip);

Last_Call_Localização=0;
}

Continuestados = dbg_Continuestados;
descanso;
Padrão padrão:
Continuestato=DBG_EXCEPTION_NOT_HANDLED; descanso;

}
descanso;
Padrão padrão:
Continuestato=DBG_EXCEPTION_NOT_HANDLED;
descanso;
}

ContinueeBogeVent (debugEvent. dwProcessId, debugEvent. dwProcessId,
ContinueeStat);
```

```
}

Caminho fechado (process_handle);
Retornar a 0;

}
```